2010

Exhaustive Statistical Analysis for Detection of Metamorphic Malware

Aditya Govindaraju
San Jose State University

Exhaustive Statistical Analysis for Detection of Metamorphic Malware

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Aditya Govindaraju

Spring 2010

ABSTRACT

Malware is a serious threat to the security of the system. With the widespread use of the World Wide Web, there has been a tremendous increase in virus attacks, making computer security an essential for every personal computer. The rat-race between virus writers and detectors has led to improved viruses and detection techniques. In recent years, metamorphic malwares have posed serious challenge to anti-virus writers. Current signature based detection techniques, heuristic based techniques are not comprehensive solutions. A formidable solution to detection of metamorphic malware is void. This paper investigates the problem of malware detection, specifically metamorphic malwares. The paper proposes a statistical based detection technique as a viable candidate for comprehensive detection of metamorphic malwares. Related work, experimental results and analysis of the results are presented in this paper.

ACKNOWLEDGEMENTS

I cannot think of anyone else to thank first, other than my advisor Dr. Robert Chun. It is hard to express in words, the magnitude of my admiration and respect for Dr. Chun. I personally feel that Dr. Chun is the best advisor anybody can get. His idea of conducting group meetings with other project students helped me in knowing perceptions of different people about my project. I am thankful to his countless number of suggestions that helped me to take my project to the finish line.

Dr. Stamp has been the primary reason for me taking up a project of this kind. Dr. Stamp inspired me to think in lines of security and I am glad I took his courses. Dr. Pollett is the first professor at SJSU made me explore and enjoy new things, through his assignments. I would like to thank Dr. Stamp and Dr. Pollett for their insights and fine tuning of my project.

I am grateful to Dr. Louden who has been extremely supportive through his timely advising whenever I was in trouble and in need of help. Dr. Horstmann always inspires me through his blogs, books, and his passion always makes me want to take that extra yard. I thank Pervez Sir for making me know what passion is and how it should be channelized.

Sri Harsha has been a friend, brother, and true well-wisher ever since I have known him. Shanthi is always tremendously supportive by showing great faith in me. I am blessed to have you both.

It is my father – Sesha Sai, mother – Satya Valli, and brother – Krishna Kanth's constant encouragement and support in whatever I wish to endeavor made me pursue Masters. I have no words to describe their love, affection, and trust.

Thank you all.

Table of Contents

List of Tables

List of Figures

1.0 INTRODUCTION

The history of computer viruses dates back to the early 1970s when ARPANET was in its early stages of adoption. Since then, computer virus infection has become one of the serious issues and posed serious threat to the integrity of computers as well as to the confidentiality of data contained within. A computer virus by definition is a program which when executed, reproduces and infects the computer, posing a threat to the integrity of the system. There are several types of computer virus namely Stealth viruses, computer worms, Trojan horses, Encrypted and polymorphic viruses, metamorphic viruses/malwares etc., A computer worm is a program, that replicates itself and spreads to other computers using the network to which the host machine is connected. Worms cause a lot of harm to the network by consuming the available bandwidth unlike viruses which affect files in a target host.

In recent years due to the widespread use of the Internet, rapid spread of virus has become possible. World Wide Web, instant messaging and file sharing systems are seen as active vulnerable nodes for spread of viruses. The nature of damage caused by viruses could be anywhere from deleting files, damaging programs to reformatting a hard disk. Some of the recent viruses aim at stealing personal information like passwords, email addresses, credit card information etc.,

Viruses are divided into two types based on the infection strategies namely Resident and Non-resident virus. A Non-resident virus has two modules namely *finder* and *replication*. The finder module is responsible for finding files to infect whereas the replication module is responsible for spreading the virus. In a resident virus, there is only a replication module which is binded to operating system calls. So, whenever the operating system performs a specified task like creating a file, the replication module gets activated.

The nature of computer viruses has evolved over the years and various anti-virus detection techniques have been developed to combat the spread of malicious viruses. There has been a lot of development both in virus detection and creating new kind of undetectable viruses. The most popular method of detecting virus is through signature detection. A virus signature is a specific sequence of bits that is unique to the virus program. It is like a fingerprint of the virus.

Signature is faster to detect and all it requires is a huge database of existing virus signatures updated periodically. However, the most recent form of viruses evades this form of detection. Such viruses are known as metamorphic virus/malware. The terms *virus* and *malware* are used interchangeably in different contexts. Metamorphic malware is a kind of malware which reproduces just like any other virus, but the new variant produced does not have the same signature. The new variant performs the same function as the parent variant but is not similar at the bit level. This implies that a single malware can produce thousands of variants. This implies that a single malware can produce the same malware with different fingerprints. Thus, the technique of signature detection fails.

This thesis aims at finding a new method for effective detection of metamorphic malware. Specifically, statistical analysis is being looked as the method of metamorphic malware detection. A malware produces a variant. A variant produces another variant malware that has the same functionality but different fingerprint. This variant produces another one and so on. However, there are certain common features amongst all the variants so produced. Firstly the functionality of the variants produced is the same, and secondly the functionality of the morphing engine. The morphing engine is responsible for generating different fingerprints of the malware. Since all variants of the malware have the same morphing engine, the side effect produced by the malware variants will be similar. This side effect is measured by calculating various statistics between the parent malware and the child variant that is produced. These statistics could include file size ratio, number of instructions and so on. It is anticipated that different malwares will possess different statistical characteristics which are unique to that particular malware. This statistical characteristic can be used as a signature to detect metamorphic malwares.

2. RELATED WORK

2.1 Malware Types

The term Malware is derived from the words *malicious* and *software.* Malwares depending on their function gather information or damage the host system without the consent of the owner. Malware includes all of the following: viruses, computer worms, trojans, rootkits, spyware, adware and other malicious software. The types of malware and the various techniques used to evade detection have changed over the years. In recent years, there has been development of new types of malware that can evade signature detection techniques used by most anti-virus softwares. These malwares are broadly classified as Polymorphic and Metamorphic malware.

2.1.1 Polymorphic Malware

Polymorphic malware like any other malware is a computer program that reproduces and causes harm to the computer. However, the variant produced by polymorphic malware constantly changes. This is done by filename changes, compression, encrypting with variable keys etc. The resulting variant has the same functionality as the parent malware. The earliest known kind of polymorphic virus was called 1260, written by Mark Washburn in 1990. The main body of a polymorphic malware consists of Malicious code and Encryption-Decryption code as shown in **Figure 1.**

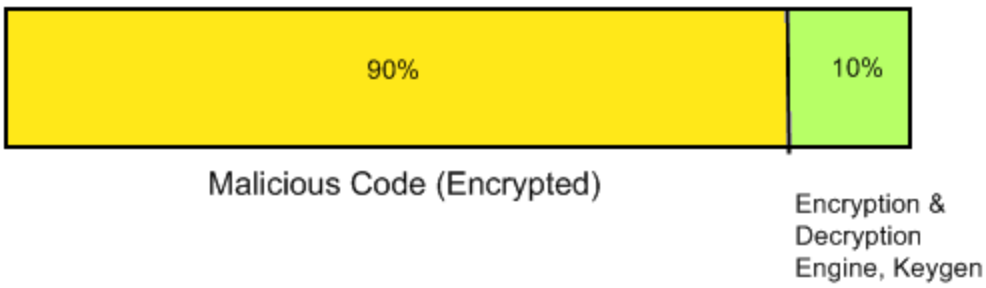

Figure 1: Anatomy of a polymorphic malware

Polymorphic malware produce different variants of itself while keeping the inherent functionality as same. This is achieved through polymorphic code. Concept of polymorphic code is core to a polymorphic malware. It is a style of code that mutates keeping the original algorithm the same. [26]

Start: GOTO Decryption_Code Encrypted: ... lots of encrypted code ... Decryption_Code: A = Encrypted Loop: B = *A B = B XOR CryptoKey *A = B A = A + 1 GOTO Loop IF NOT A = Decryption_Code GOTO Encrypted CryptoKey: some_random_number	Start: GOTO Decryption_Code Encrypted: ... lots of encrypted code ... Decryption_Code: C = C + 1 A = Encrypted Loop: B = *A C = 3214 * A B = B XOR CryptoKey *A = B C = 1 C = A + B A = A + 1 GOTO Loop IF NOT A = Decryption_Code C = C^2 GOTO Encrypted CryptoKey: some_random_number

Table 1: Polymorphic Code

The small section of polymorphic malware code containing the key generator and encryption-decryption module is responsible for morphing the malware and creating variants that do not have the same fingerprint.

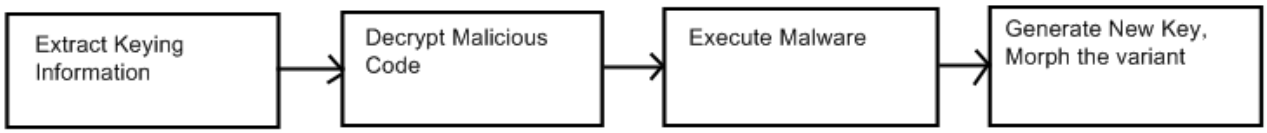

Figure 2: Execution Cycle of Polymorphic Malware

The problem of polymorphic malware is that the decryption block remained mostly the same in all the variants. The 10% of the code (as shown in Figure 1) can be used for as signature/fingerprint of the malware. Polymorphic malware is one stepping to a new generation of malwares now known as *Metamorphic malware.* A metamorphic virus magically creates variants that are entirely different in form and yet perform the same function.

2.1.2 Metamorphic Malware

Metamorphic malware represent the next class of virus that can create an entirely new variant after reproduction. The new variant produced is in no-way similar to the original variant. Metamorphic malwares do not use encryption as most polymorphic malware. Instead metamorphic malwares reply on code obfuscation techniques. Since the metamorphic malwares have do not produce variants having same body, they easily evade signature based detection. Since, most current anti-virus softwares primarily use signature based detection, metamorphic malware currently are greatest threat. [1, 2, 3, 8, 9, 10]

Unlike, polymorphic malware, metamorphic malware contain a morphing engine. The morphing engine is responsible for obfuscating the whole malware. The body of a metamorphic malware can be broadly divided into two parts namely Morphing engine and Malicious code as shown in Figure 3.

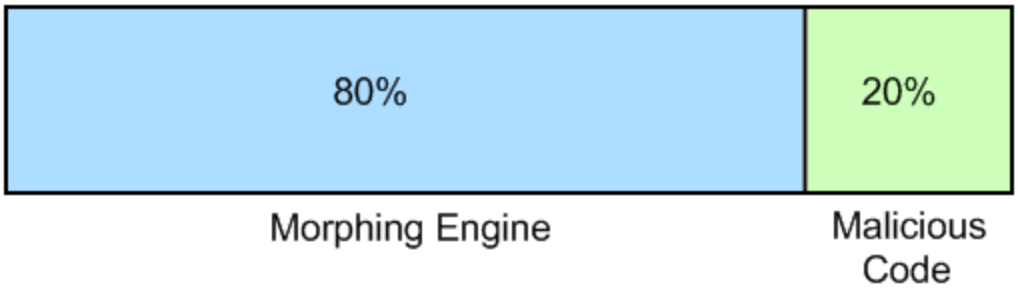

Figure 3: Anatomy of a metamorphic malware

The first known simple metamorphic virus dates back to December 1998, when Win95/Regswap virus was created. In Regswap, register exchange was implemented. Thus, different variants of Regswap had same code but different registers. [1, 2, 3, 8, 9, 10] The following code shows different versions of Regswap with the common portion in bold.

```
5A              pop  edx                    58              pop  eax
BF04000000      mov  edi,0004h              BB04000000      mov  ebx,0004h
8BF5            mov  esi,ebp                8BD5            mov  edx,ebp
B80C000000      mov  eax,000Ch              BF0C000000      mov  edi,000Ch
81C288000000    add  edx,0088h              81C088000000    add  eax,0088h
8B1A            mov  ebx,[edx]              8B30            mov  esi,[eax]
899C8618110000  mov  [esi+eax*4+00001118],ebx 89B4BA18110000 mov  [edx+edi*4+00001118],esi
```

Figure 4: Variants of Win95/Regswap virus

One of the methods of detecting regswap type of virus is using a wildcard string search. Later, virus writers used permutation techniques for reordering subroutines. The BadBoy DOS virus family relied on subroutine reordering. With 8 subroutines, the BadBoy virus could create 8! = 40320 different variants. Most of these viruses could be detected with search strings.

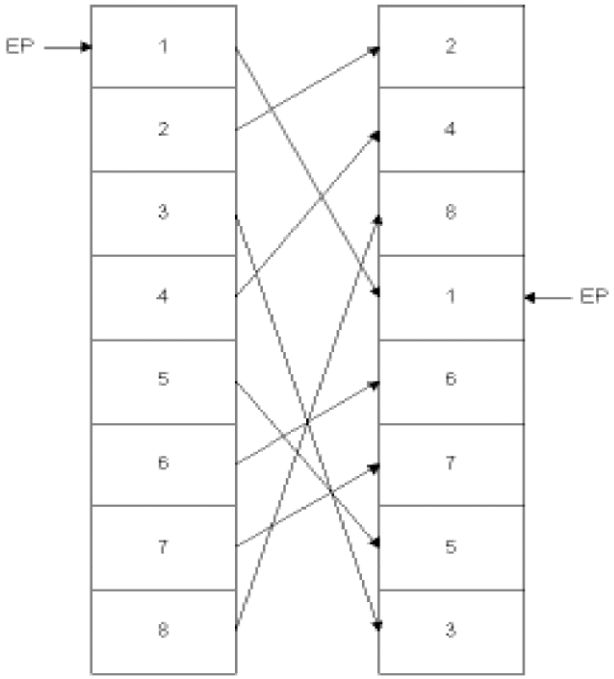

Figure 5: Module reordering

The Win32/Evol virus which appreared in 2000 embodies more complex metamorphism and permutation techniques. It contains a metamorphic engine. The following table shows the different variants of Win32/Evol Virus. The morphing engine of Win32/Evol inserts garbage instructions and uses simple code obfuscation techniques. [1, 2, 3, 8, 9, 10] Wild card based detection techniques cannot detect Win32/Evol variants.

```
BF0F000055      mov     edi,5500000Fh
893E            mov     [esi],edi
5F              pop     edi
52              push    edx
B640            mov     dh,40
BA8BEC5151      mov     edx,5151EC8Bh
53              push    ebx
8BDA            mov     ebx,edx
895E04          mov     [esi+0004],ebx
```

```
BB0F000055      mov     ebx,5500000Fh
891E            mov     [esi],ebx
5B              pop     ebx
51              push    ecx
B9CB00C05F      mov     ecx,5FC000CBh
81C1C0EB91F1    add     ecx,F191EBC0h ; ecx=5151EC8Bh
894E04          mov     [esi+0004],ecx
```

Figure 6: Win32/Evol Variants

The Morphing engine which constitutes 80% of the malware code, consists of various sub components namely Disassembler, Shrinker, Permuter, Expander, and Assembler[10].

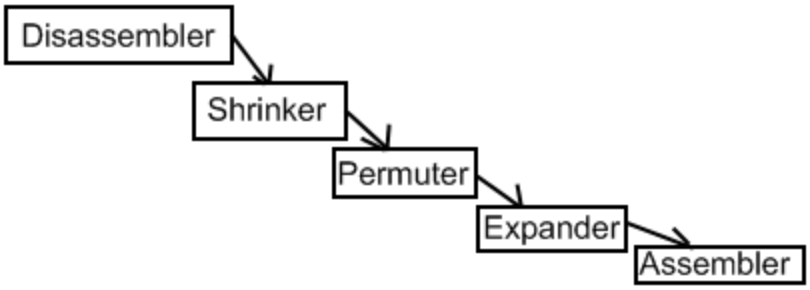

Figure 7: Parts of Metamorphic Engine

The disassembler is responsible of converting from machine language to assembly language. The engine then uses code obfuscation techniques and changes the original code into an equivalent code through code re-ordering, inserting garbage instruction, adding jump instructions etc. Obtained “code” is then permuted and shuffled using a permuter. Finally the machine code is generated using an assembler thus creating a new variant which has the same functionality but looks very much different than the parent[10].

2.2 Code Obfuscation Techniques

Metamorphic malwares use code obfuscation techniques as opposed to encryption used by polymorphic viruses. Code obfuscation is a technique of deliberately making code hard to understand and read. The resulting code after obfuscation has the same functionality. There are a variety of code obfuscation techniques namely Garbage Code Insertion, Register Renaming, Subroutine Permutation, Code reordering and Equivalent code substitution.

2.2.1 Garbage Code Insertion

In this type of obfuscation, several junk instructions are added to the program which does not affect the logic of the program. These instructions do not affect the outcome of the program. The instructions are namely XCHG, NOP, “MOV ax, ax”, “SUB ax 0” etc., These instructions make the malware look different. Also there could be blocks of code which are never called/executed. This is done in order to make it difficult for to understand the inner logic of the malware.

Original Code		**After Garbage Insertion**	
Hex	**Opcodes Assembly**	**Hex**	**Opcodes Assembly**
51	push ecx	51	push ecx
50	push eax	**90**	**nop**
5B	pop ebx	50	push eax
8D 4B 38	lea ecx, [ebx + 38h]	5B	pop ebx
50	push eax	8D 4B 38	lea ecx, [ebx + 38h]
E8	00000000 call 0h	50	push eax
5B	pop ebx	**90**	**nop**
83 C3 1C	add ebx, 1Ch	E8 00000000	call 0h
FA	cli	5B	pop ebx
8B 2B	mov ebp, [ebx]	83 C3 1C	add ebx, 1Ch
5B	pop ebx	FA	cli
		90	**nop**
		8B 2B	mov ebp, [ebx]
		5B	pop ebx

Table 2 Garbage Code Insertion

As shown the table 4, the signature of malware changes from 5150 5B8D 4B38 50E8 0000 0000 5B83 C31C FA8B 2B5B to 51**90** 505B 8D4B 3850 **90**E8 0000 0000 5B83 C31C FA**90** 8B2B 5B.However, garbage code insertion alone is not an effective method of code obfuscation. [1, 2, 3, 8, 9, 10] Virus detectors can use heuristics and threshold to detect viruses relying only on this method of obfuscation.

2.2.2 Register Renaming

In this type of obfuscation, either the name of the variables or the registers are changed. This results in different opcodes being generated. The RegSwap virus produces variants based on this principle.

Original Code	Code with Register renaming Obfuscation
MOV EAX, [X]	MOV ECX, [X]
MOV EBX, [Y]	MOV EAX, [Y]
ADD EAX, EBX	ADD ECX, EAX
MOV [X], EAX	MOV [X], ECX

Table 3: Register Renaming Obfuscation

Although the instruction set is same in both the cases, the opcodes are different. Detecting such kind of malware requires a wild card search algorithm that ignores register changes. Register renaming provides different memory traces for each variant. This makes it difficult for virus detectors. Semantically, the code is equivalent. [1, 2, 3, 8, 9, 10] Semantic based virus detectors are more useful in detecting malwares using this type of code obfuscation.

Regswap : Variant 1	Regswap : Variant 2
5A pop edx BF04000000 mov edi,0004h 8BF5 mov esi,ebp B80C000000 mov eax,000Ch 81C288000000 add edx,0088h 8B1A mov ebx,[edx] 899C8618110000 mov [esi+eax*4+00001118],ebx	58 pop eax BB04000000 mov ebx,0004h 8BD5 mov edx,ebp BF0C000000 mov edi,000Ch 81C088000000 add eax,0088h 8B30 mov esi,[eax] 89B4BA18110000 mov [edx+edi*4+00001118],esi

Table 4: Regswap variants using register renaming code obfuscation

2.2.3 Subroutine Permutation

In this type of code obfuscation the order in which the subroutines appear in the code is changed. This order is irrelevant and does not impact the functionality of the malware as the order in which a subroutine appears in the program is totally irrelevant and does not affect the execution of the program. As shown in Figure 4, the modules are re-ordered. For a malware having *n* modules, the total number of combinations is *n!* Subroutine permutation is a type of code re-ordering; however in this case the whole module is reordered rather than individual

instructions. Also some section of code inside a module could also be re-ordered [1, 2, 3, 8, 9].

Original Code	Code with Subroutine permutation
Function1:	**Function2:**
MOV EAX, [X]	MOV EBX, [Y]
Function2:	**Function1:**
MOV EBX, [Y]	MOV EAX, [X]
Function3:	**Function3:**
ADD EAX, EBX	ADD EAX, EBX
MOV [X], EAX	MOV [X], EAX

Table 5: Subroutine permutation code obfuscation

Malwares employing substitution permutation type code obfuscation can be detected through signature detection. However, detectors using techniques such as Hidden Markov Model or Profile Hidden Markov Model where the sequence of the program or subroutines is important, this obfuscation technique could pose serious problems. [1, 2, 3, 8, 9, 10] In such cases, the code needs to be de-permutated resulting in different possible combinations needing to be tried out.

2.2.4 Code Reordering through Jumps

Similar to Subroutine permutation, code reordering is a kind of code permutation. This technique uses the JMP instruction as the backbone for obfuscation. The JMP instruction is like a GOTO statement in C programming language. The execution of the program *jumps* to the position specified in the instruction. This helps in basically creating different permutations of the code while keeping the functionality constant. The number of additional JMP instructions added will be proportional to the number of lines that are re-ordered. Like register renaming, this obfuscation beats memory mapping based detection. Code reordering using jumps evades signature detection [1, 2, 3, 8, 9, 10].

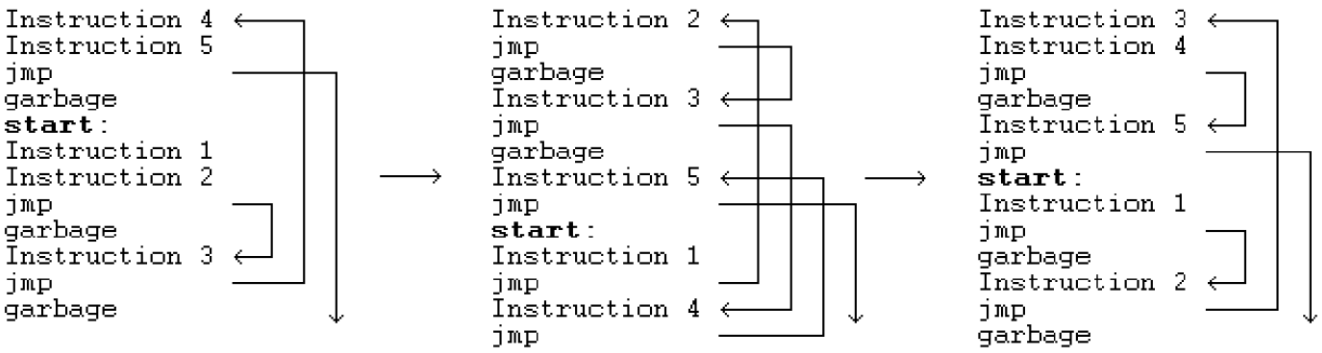

Figure 8: Code reordering using Jumps

2.2.5 Equivalent code substitution

This technique of code obfuscation relies on the fact that an operation can be done in many ways. This principle is fundamental to mathematics and also in Boolean circuits. An XOR gate as shown in Figure 7 has different variants. All variants are functionally the same, but appear different. [1, 2, 3, 8, 9, 10] This is one of the dangerous forms of code obfuscation as it completely changes the topology and quality of variants produced.

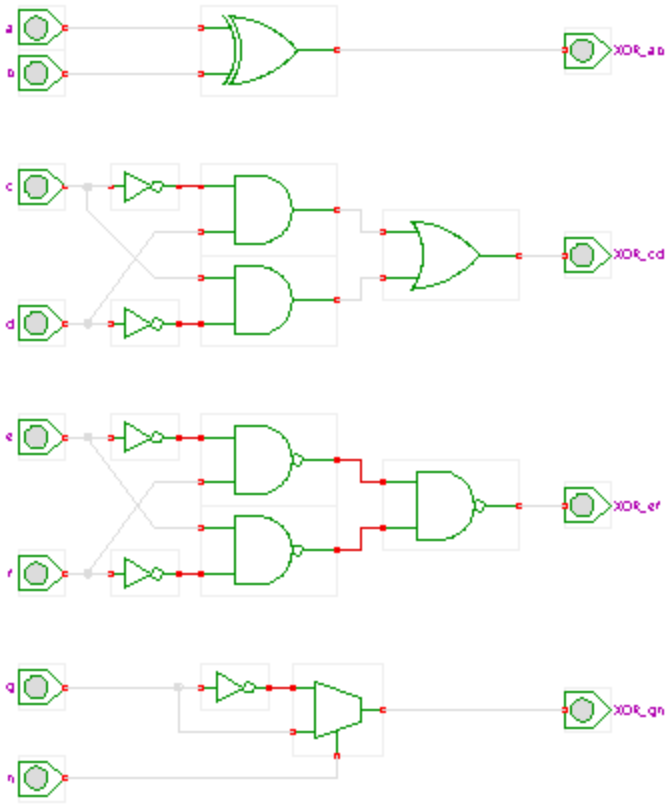

Figure 9: Three different variants of XOR. Functionally same, physically different

This kind of code obfuscation technique evades most of the malware detection techniques. Behavior based malware detectors work best in this case as the functionality remains the same for all variants.

Original Code	Code with Equivalent code obfuscation

MOV EAX, [X]
MOV EBX, [Y]
ADD EAX, EBX
MOV [X], EAX
	XOR EAX, EAX
	ADD EAX, [X]
	ADD EAX, [Y]
	MOV [X], ECX

Table 6: Equivalent code obfuscation

2.3 Existing Metamorphic Malware Detection techniques

2.3.1 Signature based detection

Signature based detection is one of the most popular and effective way of detecting virus. A signature as the name suggests is an identification. It is like a person's fingerprint. It is used to identify a single virus or a family of virus. Signature based detectors need to update its list of signatures frequently. If a new virus is found, then the signatures present in the database would be of no use since the signature of the new virus is not available in the database. Although anti-virus detectors do not completely rely on signatures, they also look into heuristics sometimes. However, signatures form a formidable percentage in the detection process. Signature detection is fast and simple.

Signatures detection is not effective against some polymorphic and most metamorphic viruses. Metamorphic malwares use a combination of various code obfuscation techniques making the resulting variant so similar to the parent malware. Since signature detection depends on a series of bits, each variant has a different set of bits. Storing signatures of each variant of a malware is practically not feasible since it increases the dictionary of the detector with unnecessary signatures.

2.3.2 Heuristics based detection

Recent virus detectors use signature detection along with heuristics. This help in reducing the number of false alarms. Heuristics is partially dependent on the behavior of the target malware. Each virus is born with a purpose. Not all virus have the same purpose. The technique, methodology and the kinds of attacks used by each virus is unique of its kind. Heuristics sets

certain base rules. These rules are mostly essential for proper functioning of the system and maintain its stability along with data integrity. Since a virus aims at tarnishing one or many of these, heuristics help in detecting and preventing new viruses to some extent.

Operating system files, DLL files (in Windows Operating System), registry entries, networking are some of the important places which are prone to attack. If a file or program is found to be violating the rules of normal program behavior then that program can be suspected to be a virus.

Although Heuristics is promising, it cannot stand for itself. The number of false alarms raised is high. Heuristics is good when combined with another reliable detection technique. Heuristics can be used to support or augment results of other detection techniques. A virus coder can still write a virus that does not obey the rules in a heuristics and still cause considerable damage to the system.

2.3.3 Behavioral based detection

Behavior based detection is a type of dynamic analysis techniques. In a dynamic analysis technique the malware under suspicion needs to be dynamically executed in a sand-box environment. In behavior based detection, the malware is executed and the behavior of the malware is analyzed in runtime. Whenever the behavior of the virus seems “bad”, it is flagged as virus and corrective action is taken. Behavior based detection requires “templates” of bad/suspicious behavior. The behavior of the malware becomes its signature. Thus behavior based detection technique is a kind of signature based detector except that the signature here is the functionality of the malware. Since most recent malwares use code obfuscation techniques, it is practically not possible to statically analyze the program's functionality.

Behavioral based detection would result in less false positives. It is more reliable. Virus has to be debugged in runtime. This detection mechanism requires templates of bad behavior which can be automatically created or updated. It requires a dynamic environment which emulates the operating system and yet not harm the system. Virtual machines can be used as test bed for running malwares.

2.3.4 Semantic based detection

This detection technique is based on static analysis. In static analysis, the malware is not executed. The malware code is used to determine the malicious nature of the code. A signature is created based on the semantic property of the code. [25]

$$\text{Signature } T = (I_T, V_T, C_T)$$

I_T = Sequence of Instructions

V_T = Set of variables

C_T = Set of Symbolic constants.

Semantic based detection supports certain types of code obfuscation techniques such as Instruction reordering, Register renaming, Garbage Insertion. However, malwares implementing Instruction replacement and Equivalent code replacement can still evade semantic based detection. [25]

2.3.5 Hidden Markov Model based detection

In recent years, detection of metamorphic malware has been very much effective due to the application of Markov models to malware detection. Profile Hidden Markov Models (PHMM) is known for their success in determining relations between DNA and protein sequences. When applied for malware analysis it has been found that PHMM can effectively detect metamorphic malware. Although PHMM can detect malwares which are metamorphic, they still need a test data in order to train them. [1] Also, the process of filtering the data, disassembling them, training and scoring the whole dataset can be time consuming.

2.3.6 Similarity Analysis

Another method to detect malwares is through similarity analysis. A program is represented as some number of functions f, and each function contains some number of statements which are termed as vectors x and y. The total number of vectors for the same program P and for all functions f is kept same. Similarity analysis can be performed by using cosine similarity measure which is primarily used in text mining. In short, the maliciousness of a

code is estimated.[5]

The following steps are involved in a typical similarity analysis based detection. The program executable is decompressed and disassembled. Each disassembled program represents a vector of functions. Each function is represented as an array of vector of functions. The similarity between the functions of a program P and P' is computed using cosine similarity measure or other methods. The value is them compared with the threshold value to determine if the given executable is malicious or not[5].

3.0 PROPOSED METAMORPHIC MALWARE DETECTION TECHNIQUE

3.1 Why Statistics?

Although it is important to detect metamorphic malwares at the first hand, it is equally important that this detection is done at a reasonable speed, performance and accuracy. The above mentioned methods lack either one or all of these qualities. The main aim of the project is to use statistics as the key ingredient to detecting metamorphic malwares. Since extracting statistics from assembly code is very fast, speed and performance would not be an issue if this method succeeds. However, the accuracy of this mechanism must be at par with the above mentioned methodologies.

3.2 Current Research

Statistical based malware detection is a new technique developed in the past couple of years. It is a fairly new domain in malware detection arena. Andrew Wallenstein et al [18] proposed that Statistics derived from code analysis of different variants of same malware can be used as a signature. This signature can then be used to detect metamorphic malwares. Statistical based malware detection is dynamic. So, the malware needs to be executed in a sand-box environment.

Statistical based detection does not look at one variant of malware. It looks at generations of variants of a single malware by executing them multiple times. The base version of malware is executed. This produces another variant. This variant is fed as input and another malware is produced and so on. Since malwares of same variant have the same morphing engine, the kind of code obfuscation produced will be similar. But this similarity is not clearly evident. It is hidden in the form of statistics. Applying statistical analysis over a generation of malware variants will help in developing a process for automatic signature generation for a class of malware.

An exhaustive list of statistics is extracted from each generation of malware variant. Some of statistics include file size increase, file size per instruction, garbage code metrics, compression etc. Not all of these statistics are necessary for all types of malware variants. This will lead to the development of a classifier based detection engine that will look into particular statistical values. A threshold value needs to be ascertained individually for each type of malware

family. These exhaustive statistics coupled with threshold values and classifier should help in comprehensively detecting metamorphic malwares.

3.3 Proposed Method in detail

The current research in malware detection goes under the following paradigm. The paradigm is to detect malware from normal executable. Since a malware employs the following techniques such as Garbage code Insertion, Code reordering, Register renaming, Alternative Instruction substitution, the current techniques highlight on the side effects that are produced due to these techniques.

However, there is another paradigm which is ignored. Our proposed model is based on this paradigm. The proposed model is based on the paradigm of **optimization**. What is the difference between a normal executable and a malware? A normal executable is generated by a compiler. Thus, all the instructions that the compiler produces in EXE file is optimized. A malware file on the other hand is initially generated by a compiler. But, the next generations of the malware is not generated by the compiler. It is generated by the malware itself. Thus, it is not as **optimized** as a normal executable. Our aim is indirectly to find out how optimized a program is with respect to the normal programs and then decide if the program is malware or not.

We disassemble the exe files of a large dataset preferably around five hundred or more executables. Once disassembled, statistics are extracted from them. Similarly a dataset of more than hundred different malware variants are disassembled and statistics are generated. A threshold value is found out which will help us in deciding whether an executable is malware or not.

4.0 DESIGN

4.1 Overview

Traditionally viruses and other malicious programs are detected using various signature based methods and other heuristics. However, in case of metamorphic malwares, these techniques are ineffective. Thus a need for a new detection mechanism for detecting metamorphic malwares exists. The following project starts with the assumption that statistics can be used to detecting metamorphic malwares. Although the body of the malware changes, they are still generated by the same engine. Therefore, a common trait or pattern should exist amongst all variants of the malware no matter how different the variants are from one another. The proposed methodology tries to prove the assumption that statistics can be used as a tool to find this common trait or pattern visible in all variants of a family of malware.

The proposed methodology consists of 3 major components namely disassembler, statistics extractor and C4.5 classifier. They are described in detail in the following sections.

4.2 Disassembler

A disassembler is a computer program which takes an executable as input and returns the assembly language code of it. Its function is exactly the opposite of an assembler program found in compilers. Converting into assembly language helps in getting an insight into the source code of the program. No matter in what language a program is written, once an executable is created, it can be converted into assembly source code using disassemblers.

4.3 Statistics Extractor

The statistics extractor is the heart of the proposed detection methodology. This module takes the assembly code generated by a disassembler as input and analyzes the source code. It then gathers important statistics from this source code. This statistics finally help in deciding whether a given executable is malware or not.

The statistics extracted are detailed as below. The statistics are selected in a way so as to

counter act the code obfuscation techniques generally adopted by malware writers.

1. Percentage of NOPs at the end of Sub-routine (PER_NOP_AT_END_SUB)

$$\frac{(100 * \text{NOP instructions which are at the end of a sub-routine})}{(\text{Total number of assembly LOC})}$$

One of the common methods of code obfuscation employed by malware writers is Garbage Code insertion. In this type of obfuscation, certain instructions are inserted which do not affect the logic of the program. An NOP instruction is an example and is the most common and widely used instruction by malware writers. The following statistics finds out the percentage of NOP instructions which are present at the end of the sub-routine. The reason for using this statistics is that, traditionally NOPs are not frequently used in a legitimate exe. Now-a-days with the advancement of compiler optimization, the usage of NOP's has decreased in legitimate executables. In addition, NOPs are generally found at the end of a sub-routine. In short, it means that an exe with higher concentration of NOPs at the end of subroutine is less likely to be a legitimate exe.

2. Percentage of NOPs at Random (PER_NOP_AT_RANDOM)

$$\frac{(100 * \text{NOP instructions which are NOT at the end of a sub-routine})}{(\text{total number of assembly LOC})}$$

This statistics gives the percentage of NOPs which are placed at random. This is a very powerful statistic for detecting malwares because metamorphic malware engines place NOPs at random. So, a metamorphic engine which heavily uses garbage code insertion should be easily captured by this statistic.

3. JMP Instruction profile (JMP_PROFILE_ALL)

$$\frac{(100 * \text{count of all 32 JMP instruction variants})}{(\text{Total number of assembly LOC})}$$

Another common method of code obfuscation is called code re-ordering. In this type of obfuscation the code of a subroutine is initially split. A JMP instruction is inserted in between to

join these two codes. Therefore, the logic of the code is not changed whereas the body of malware changes.

The following statistic finds out how many times a JMP instruction is called with respect to the number of lines of assembly code. Dividing by number of lines of code helps in normalizing the statistic. This statistic helps in effectively finding out metamorphic malwares engines using code reordering.

4. SHORT JMP Instruction profile (SHORT_BY_JMP_PROFILE)

This is similar to the previous statistics, except that in this case the count is gathered for all JMP instructions using the SHORT profile.

$$\frac{(100 * \text{count of all 32 JMP instruction variants using SHORT })}{(\text{Total number of assembly LOC})}$$

5. SUB_ROUTINE_PROFILE_ALL

$$\frac{[\ (\text{Total number of sub routines defined}) + (\text{Total number of locations defined})\ ] * 100}{(\text{Total number of assembly LOC})}$$

This set of statistics helps in detecting a kind of code obfuscation named Sub-routine reordering. In this case, the presence of sub-routines in the assembly code is physically changed from one place to another resulting in a different malware body. Locations are labels where a segment of code is defined.

6. SUB_MINUS_CALL

$$\frac{[\ (\text{total number of sub routines defined}) + (\text{total number of locations defined}) - (\text{total number of CALL instructions}) - (\text{total number of all 32 variants of JMP instructions})\ ] * 100}{\div (\text{Total number of assembly LOC})}$$

This statistic helps in detecting code obfuscation where dummy sub-routines are randomly added to the malware body. These subroutines are never called. The following statistics helps in finding out how if there are any sub-routines which are left dummy and never called.

4.4 C4.5 Classifier

Once the statistics are extracted for normal and malware executables there is a need to develop a decision tree based on the statistical values obtained from test data. This decision tree is obtained using C4.5. C4.5 was originally developed by Ross Quinlan. It is an algorithm for generating decision trees. It is a statistical classifier.

C4.5 builds decision trees based on information entropy. A test data contains a set of samples S= S_1, S_2…S_n Since it is training data, it is already classified into different classes C=C_1, C_2….C_n Using this data, C4.5 finds out attributes which effectively split the given sample data from one class to another.

4.5 The Process

The whole process of the proposed methodology is described in the following figure.

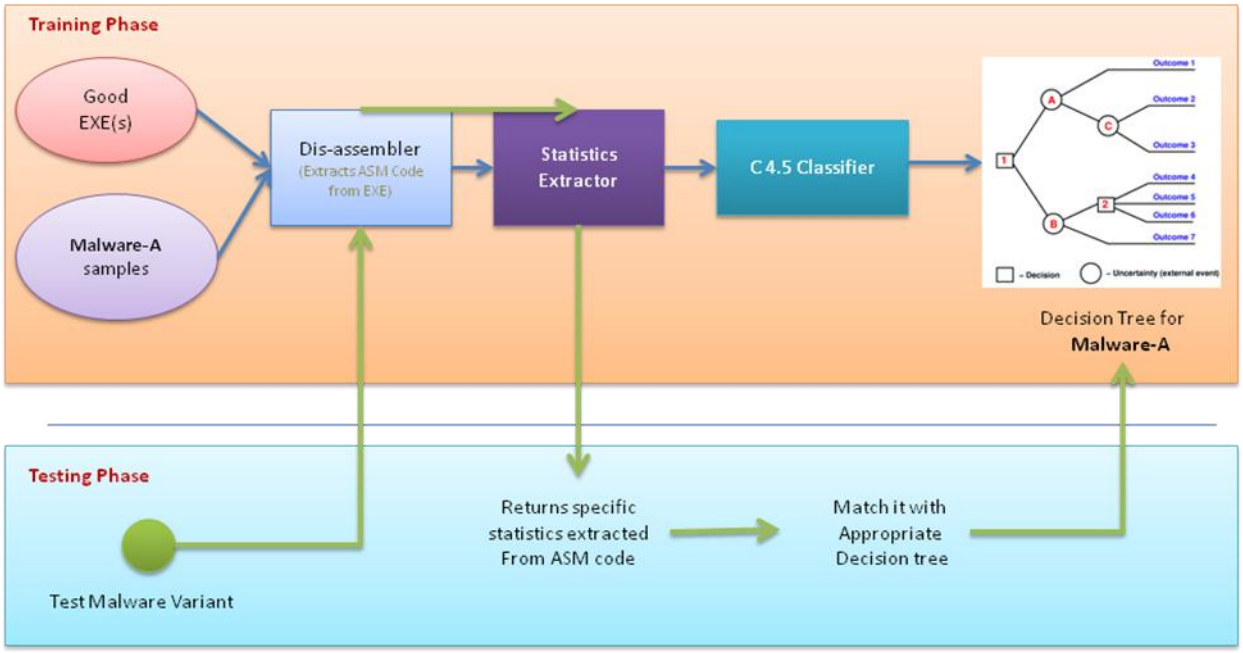

Figure 10: Proposed Technique in detail

Initially a set of good exes are chosen. They are sent into a disassembler and assembly code is extracted from them. In addition, a set of previously chosen metamorphic malware samples are disassembled. Then, they are sent through the statistic extractor module which extracts the above mentioned statistics from assembly source code. The test data is then passed through a C4.5 classifier which gives out the decision tree.

When a test exe is found, it is disassembled and passed through the statistic extractor. Then, the values obtained are matched to see, if it fits the decision tree of any existing malware family. If it fits, then the new test exe is termed as a variant of that malware, else it is declared benign.

5 TEST CASES

5.1 Training the Classifier

C4.5 is a statistical classifier. In order to churn decision trees out of C4.5, it needs a sample data set to understand the way normal and malwares are classified. A clean Windows XP system was chosen and about 500 executables were randomly taken. These 500 executables included the standard windows operating system applications and some third party executables that come pre-installed with the system. These 500 executables were passed through the proposed system and statistics were extracted. The following figure gives the distribution of the training data for normal/genuine executables.

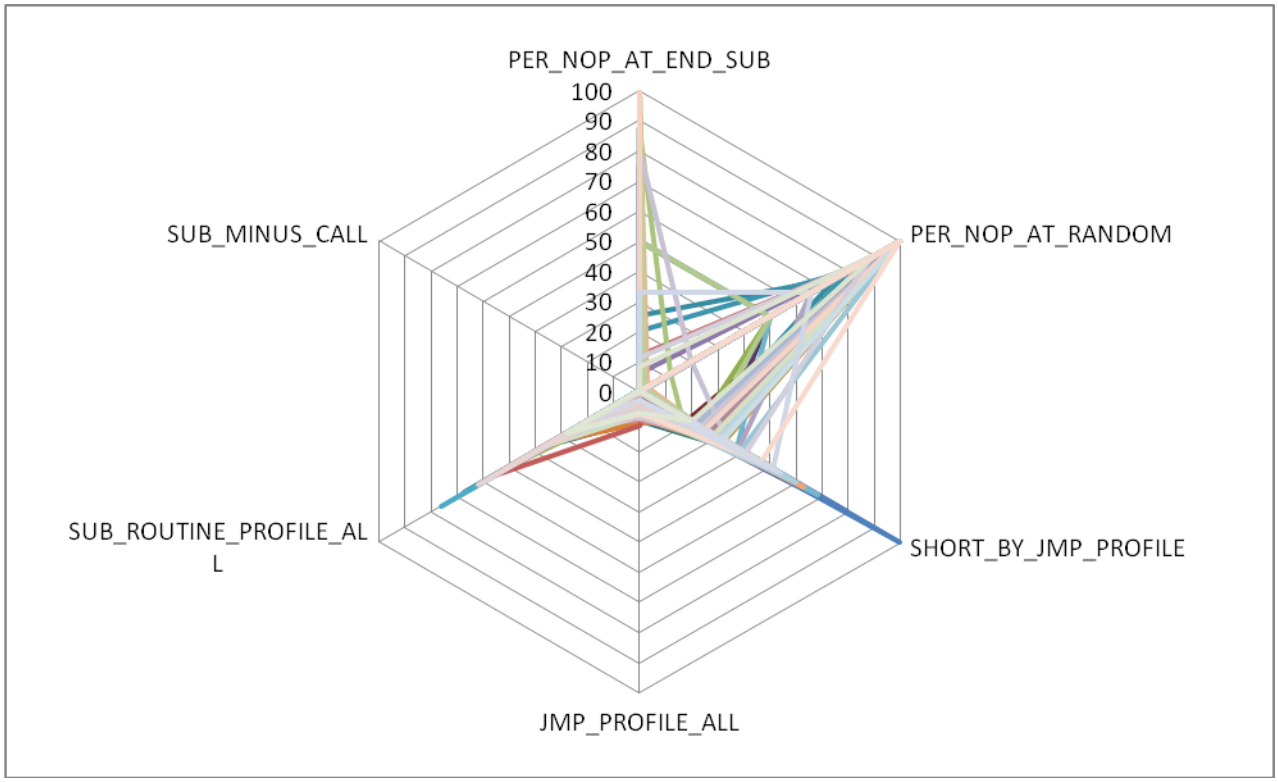

Figure 11: Footprint of Normal/Genuine Executables Training data topology

5.2 Next Generation Virus Construction Kit

NGVCK is a computer program which can be used to generate different kinds of metamorphic malware. It is one of the best known metamorphic malware generators. Initially 100 variants of NGVCK are generated and 7 variants are picked at random. Similarly 500 normal exe from a clean windows xp operating system are taken. They form the test dataset for NGVCK malware. The test data is disassembled and passed through the statistic extractor. The following

figure gives the statistical distribution of test NGVCK data.

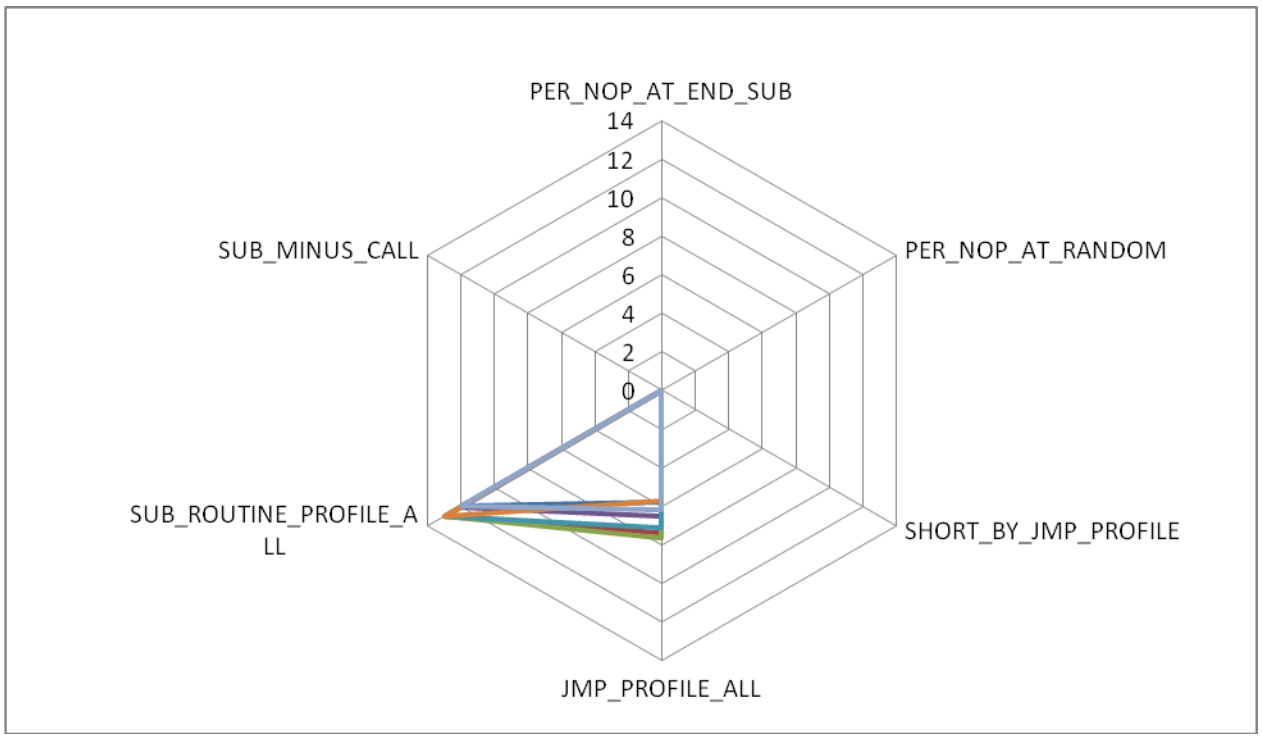

Figure 12: Footprint of NGVCK Training data topology

The test data set is passed through C4.5 classifier to generate a decision tree for NGVCK variants. The following tree is obtained.

SUB_MINUS_CALL > 0.055 : normal (491.0)

SUB_MINUS_CALL <= 0.055 :

| SHORT_BY_JMP_PROFILE <= 9.333 : **malware** (7.0)

| SHORT_BY_JMP_PROFILE > 9.333 : normal (18.0)

Now, the remaining 93 variants of NGVCK were disassembled and statistics were extracted from them.

JMP_PROFILE_ALL	SUB_ROUTINE_PROFILE_ALL	SUB_MINUS_CALL
5.993	11	0.004
6.349	12	0.012
7.184	13	0.01
7.573	13	0.012
6.695	12	0.006
7.495	13	0.006
6.715	12	0.015
5.95	13	0.021
6.589	13	0.019
5.847	11	0.01
7.753	12	0.002
6.144	12	0.008

Table 7: NGVCK Test Data results sample

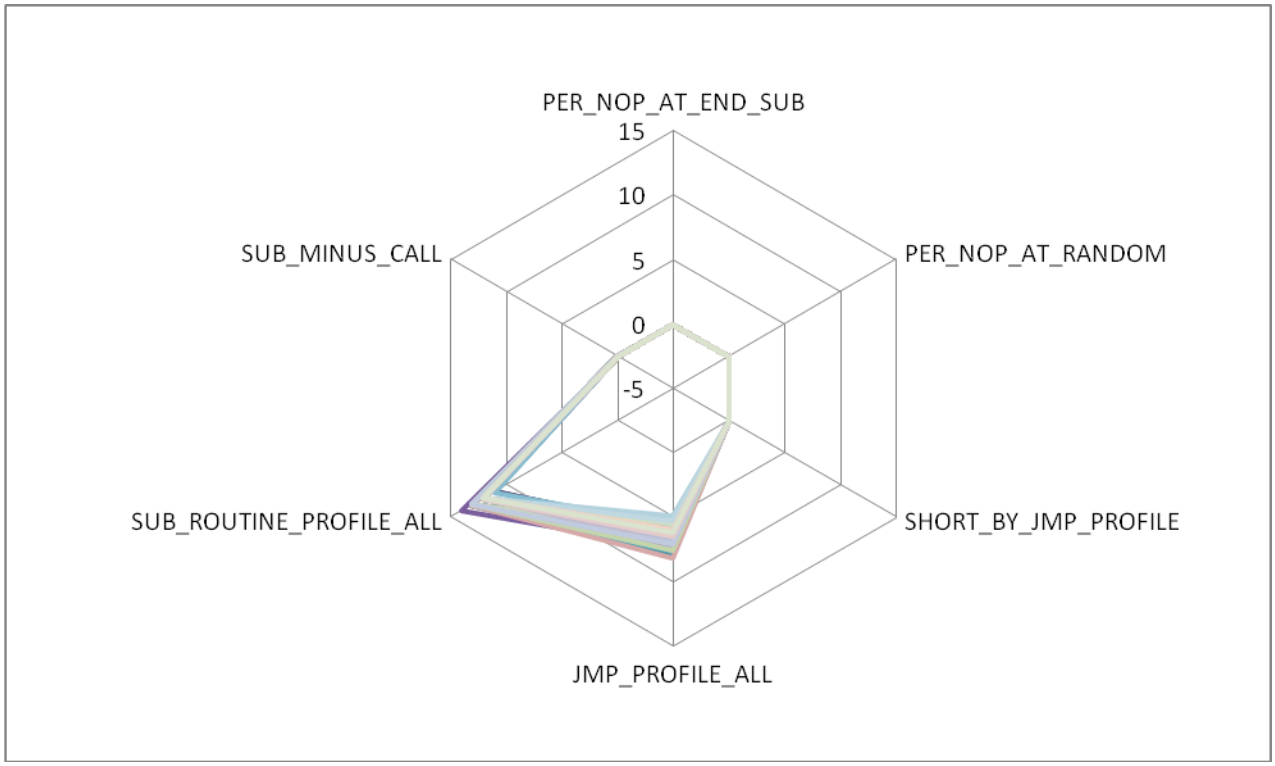

Figure 13: Footprint of NGVCK Test data results and topology

All of the 93 NGVCK variants had

- SUB_MINUS_CALL <= 0.055 and
- SHORT_BY_JMP_PROFILE <= 9.33 which is in consonance with the decision tree for NGVCK malware family variants.

5.3 Virus Creation Lab (VCL)

Virus Creation Lab is a computer program which can be used to generate different kinds of metamorphic malware. Initially 9 variants of VCL are generated and 5 variants are picked at random. Similarly 500 normal exe from a clean windows xp operating system are taken. They form the test dataset for VCL malware. The test data is disassembled and passed through the statistic extractor. The following figure gives the statistical distribution of test VCL data.

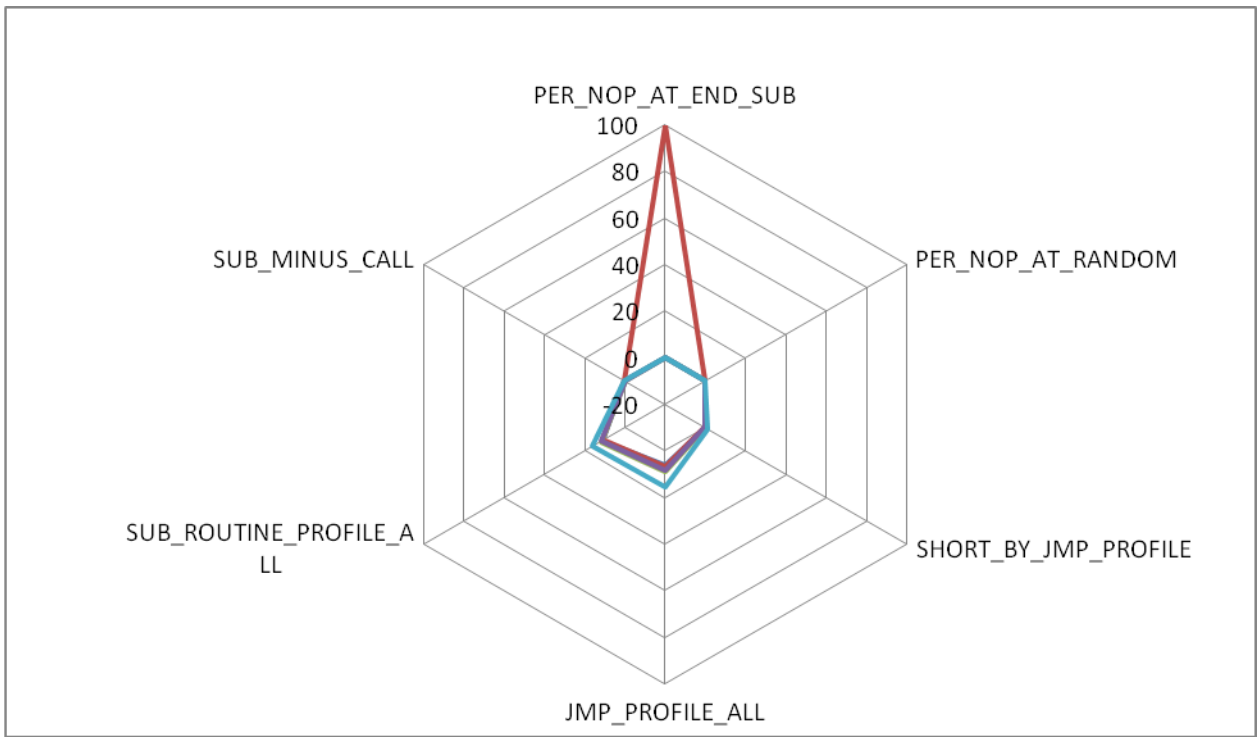

Figure 14: Footprint of VCL Training data topology

The test data set is passed through C4.5 classifier to generate a decision tree for VCL variants. The following tree is obtained.

SUB_MINUS_CALL > 0.056 : normal (489.0)

SUB_MINUS_CALL <= 0.056 :

| SUB_MINUS_CALL <= -0.041 : **malware** (5.0)

| SUB_MINUS_CALL > -0.041 : normal (20.0)

Now, the remaining 4 variants of VCL were disassembled and statistics were extracted from them.

JMP_PROFILE_ALL	SUB_ROUTINE_PROFILE_ALL	SUB_MINUS_CALL
5.183	11	-0.043
15.425	16	-0.085
6.498	12	-0.043
9.063	12	-0.041

Table 8: VCL Test Data results sample

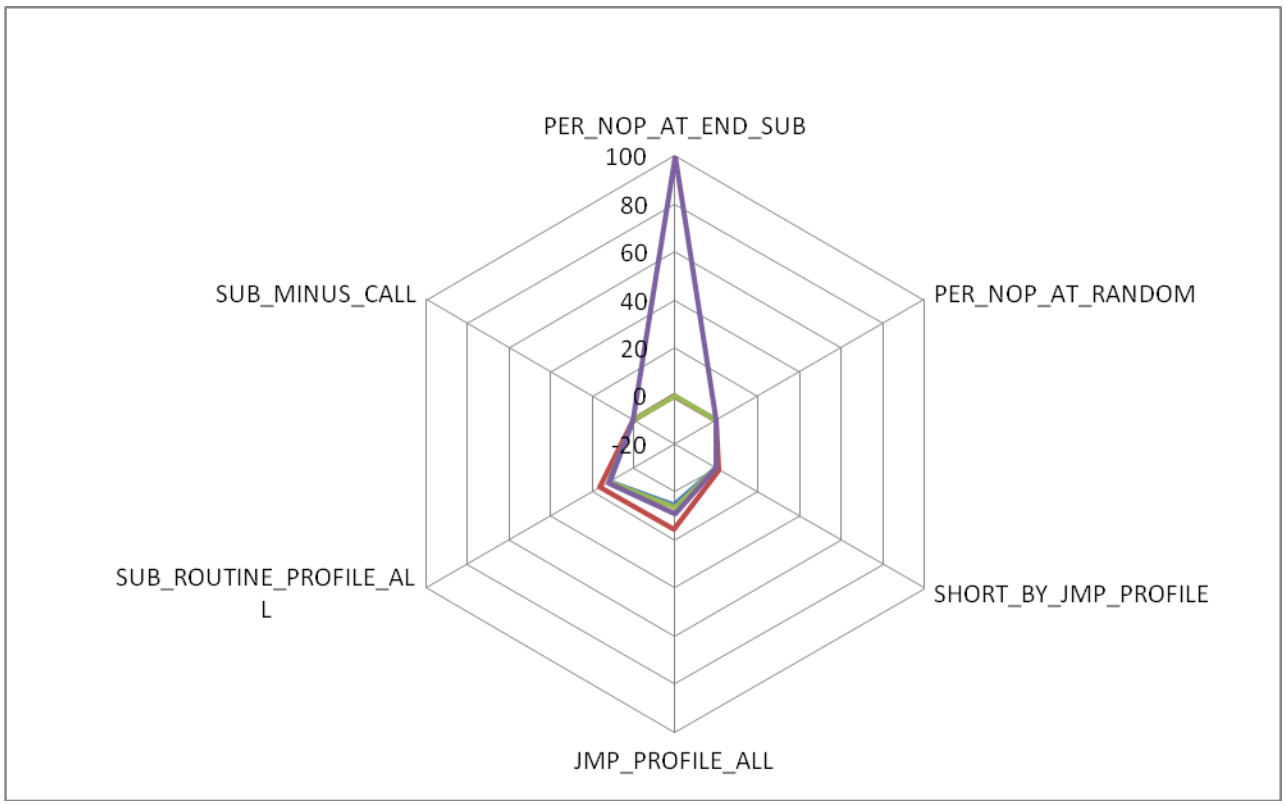

Figure 15: Footprint of VCL Test data results and topology

All of the 4 VCL variants had SUB_MINUS_CALL <= -0.041 which is in consonance with the decision tree for VCL malware family variants.

5.4 G2

G2 is a computer program which can be used to generate different kinds of metamorphic malware. Initially 6 variants of G2 are generated and 3 variants are picked at random. Similarly 500 normal exe from a clean windows xp operating system are taken. They form the test dataset for G2 malware. The test data is disassembled and passed through the statistic extractor. The following figure gives the statistical distribution of test G2 data.

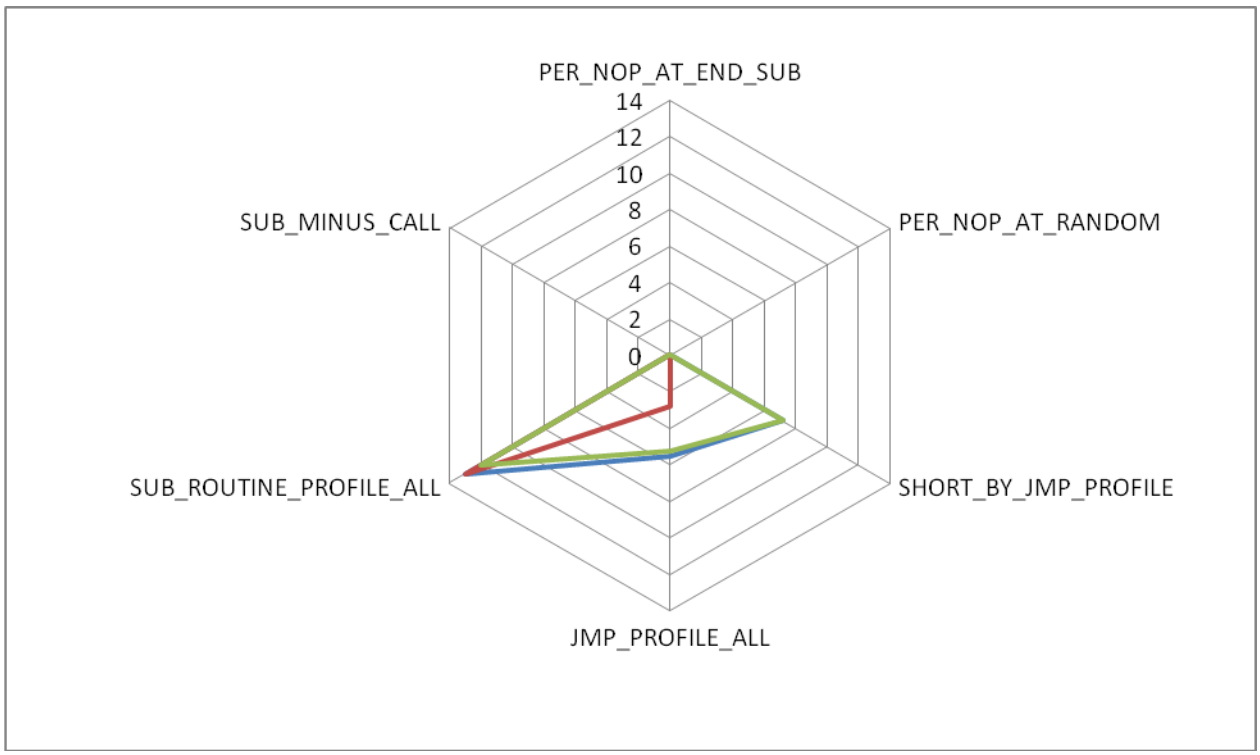

Figure 16: G2 Training data topology

The test data set is passed through C4.5 classifier to generate a decision tree for G2 variants. The following tree is obtained.

SUB_ROUTINE_PROFILE_ALL > 20 : normal (477.0)

SUB_ROUTINE_PROFILE_ALL <= 20 :

| SHORT_BY_JMP_PROFILE <= 7.143 : malware (3.1/0.1)

| SHORT_BY_JMP_PROFILE > 7.143 : normal (31.9/0.0)

Now, the remaining 3 variants of G2 were disassembled and statistics were extracted from them.

STATS_JM	SHORT_BY_JMP_PROFILE	JMP_PROI	SUB_ROUTINE_PROFILE_ALL	SUB_MINU
2	0	6.742	13	0.056
1	7.143	5.469	13	0.062
2	7.143	5.882	13	0.063

Table 9: G2 Test Data results sample

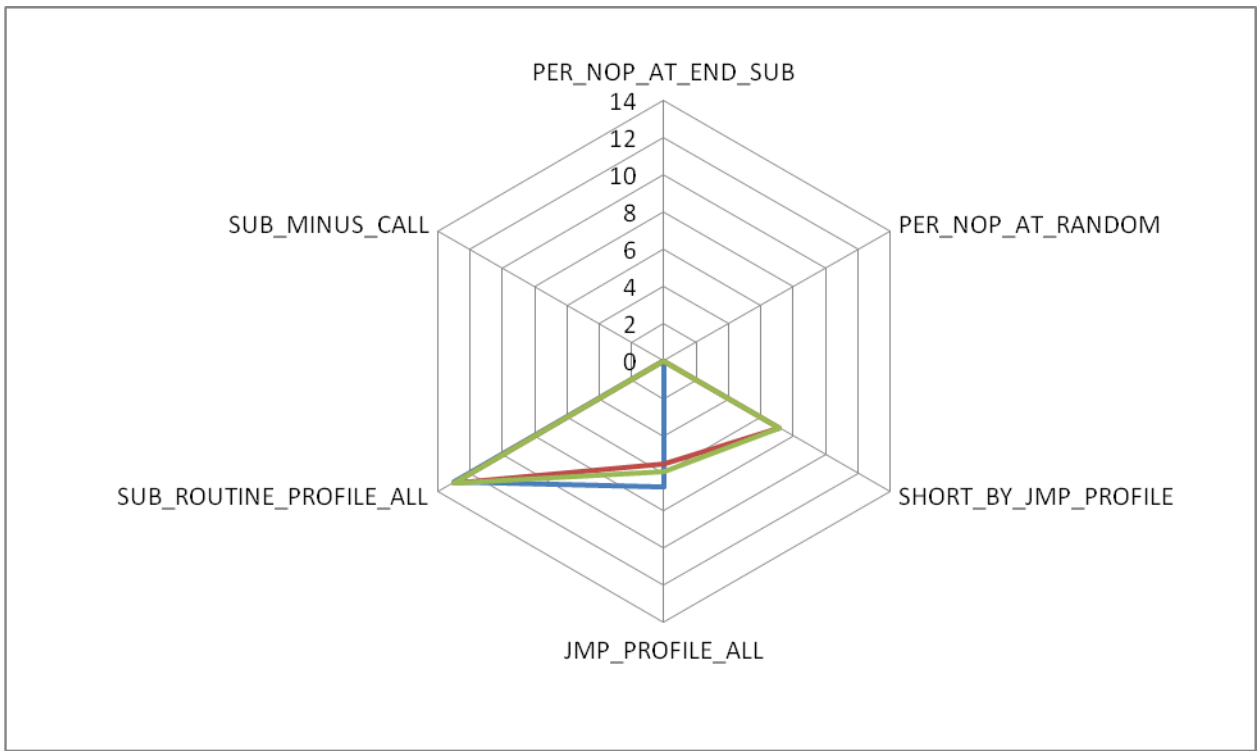

Figure17: G2 Test data results and topology

All of the 3 G2 variants had SUB_ROUTINE_PROFILE_ALL <= 20 and SHORT_BY_JMP_PROFILE <= 7.143 which is in consonance with the decision tree for G2 malware family variants.

5.5 MPCGEN (Mass Code Generator)

MPCGEN is a computer program which can be used to generate different kinds of metamorphic malware. Initially 20 variants of MPCGEN are generated and 5 variants are picked at random. Similarly 500 normal exe from a clean windows xp operating system are taken. They form the test dataset for MPCGEN malware. The test data is disassembled and passed through the statistic extractor. The following figure gives the statistical distribution of test MPCGEN data.

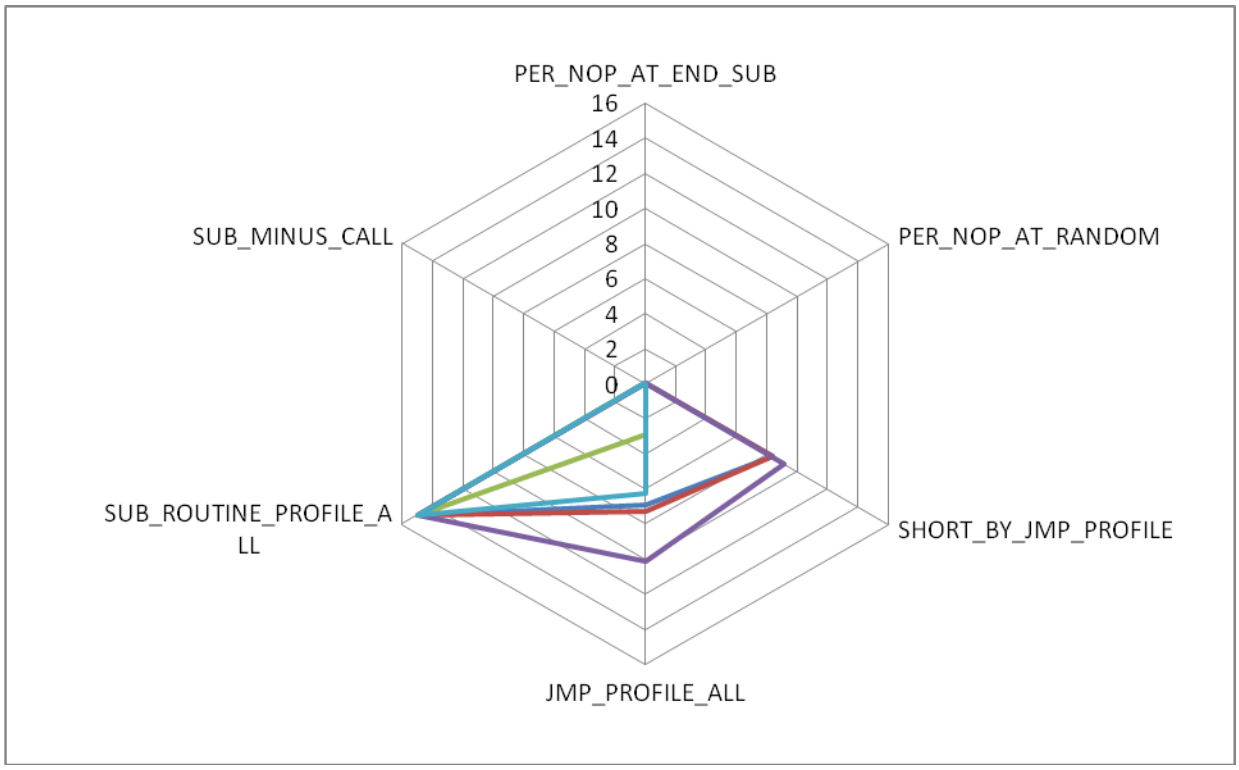

Figure18: MPCGEN Training data topology

The test data set is passed through C4.5 classifier to generate a decision tree for MPCGEN variants. The following tree is obtained.

SUB_ROUTINE_PROFILE_ALL > 20: normal (477.0)

SUB_ROUTINE_PROFILE_ALL <= 20:

SHORT_BY_JMP_PROFILE > 19.054: normal (26.7)

SHORT_BY_JMP_PROFILE <= 19.054:

SUB_MINUS_CALL <= 0.116: malware (7.0)

SUB_MINUS_CALL > 0.116: normal (5.3/0.0)

Now, the remaining 15 variants of G2 were disassembled and statistics were extracted from them.

STATS_JM	SHORT_BY_JMP_PROFILE	JMP_PROFI	SUB_ROUTINE_PROFILE_ALL	SUB_MINUS_CALL
3	9.091	6.509	15	0.065
2	7.143	5.833	15	0.088
4	8.333	9.756	15	0.033
2	0	7.303	14	0.051
2	0	5.164	15	0.085
2	6.25	5.556	14	0.08
3	17.647	6.071	16	0.075
4	8.333	8.108	14	0.041
2	6.25	6.426	14	0.072
2	0	5.34	15	0.078
4	8.333	8.108	14	0.041
3	8.333	6.091	15	0.066

Table 10: MPCGEN Test Data results sample

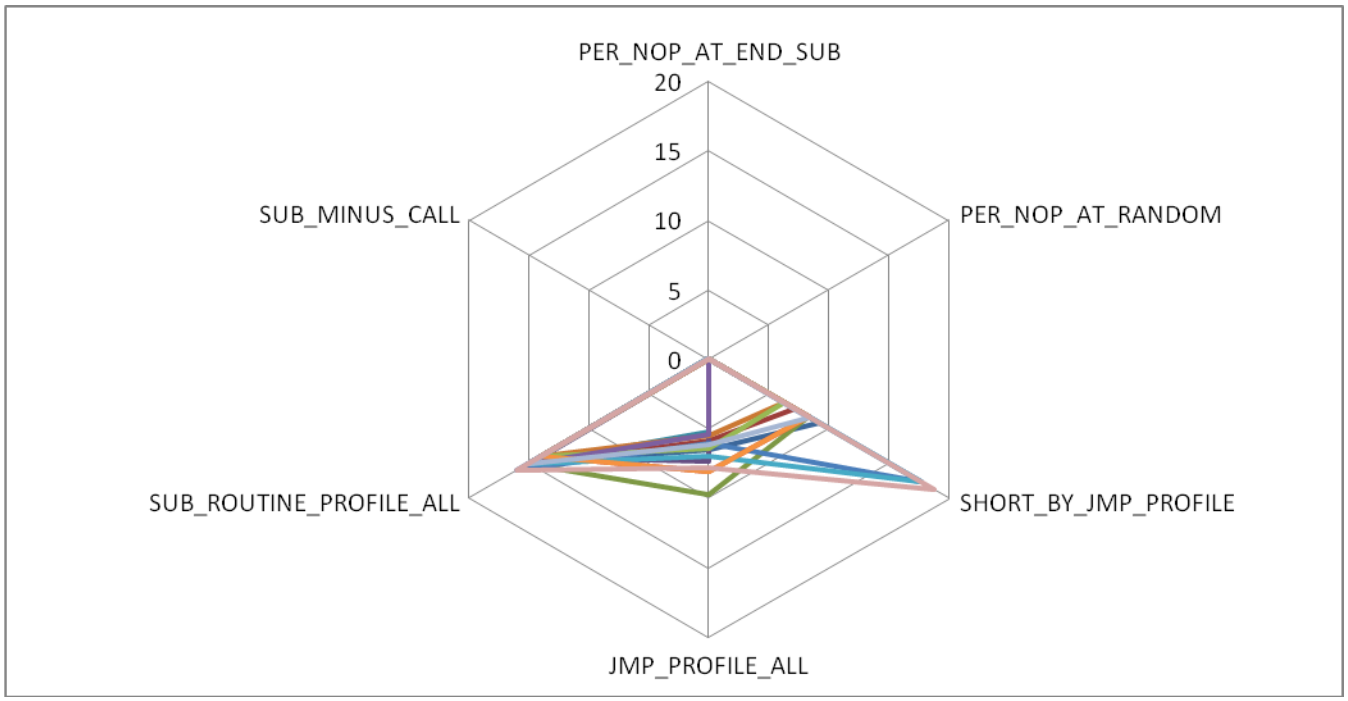

Figure 19: MPCGEN Test data results and topology

All of the 15 MPCGEN variants had SUB_ROUTINE_PROFILE_ALL <= 19.054 and SUB_MINUS_CALL <= 0.116 which is in consonance with the decision tree for MPCGEN malware family variants. Table 10 shows the test data for all fifteen variants and Figure 18 shows the footprint of MPCGEN test data.

5.6 Win32/Zmist

The Win32/Zmist virus is a very complex metamorphic virus released in the year 2001. The abbreviation Zmist stands for Zombie Mistfall. Initially 3 variants of Zmist were found, out of which 2 were selected at random for training purposes. Similarly 500 normal exe from a clean windows xp operating system are taken. They form the test dataset for Zmist malware. The test data is disassembled and passed through the statistic extractor. The following figure gives the statistical distribution of test ZMIST data.

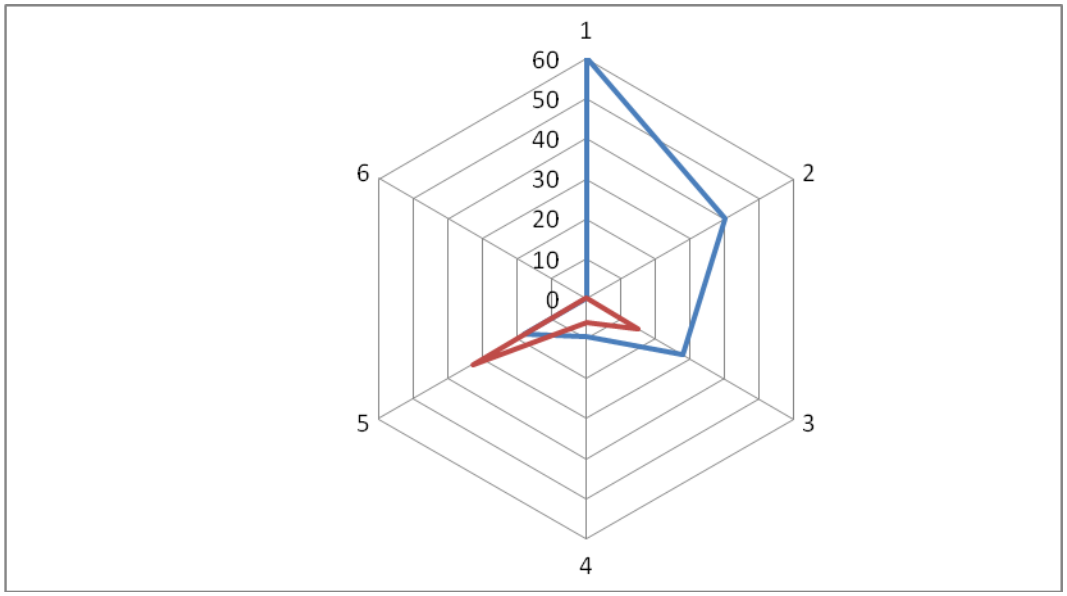

Figure 20: ZMIST Training data topology

As seen above, the distribution of test data of ZMIST virus is very random and does not follow any pattern when compared to other malwares like that NGVCK, VCL, G2, and MPCGEN. When the training data was fed to C4.5, it could not produce any decision tree of the given data.

This implies that the current statistics are not enough to capture ZMIST. The ZMIST virus used advanced code obfuscation like reversing of branch conditions, push/pop replacing register movement, encoding alternative opcodes, xor/sub and or/test interchanging and garbage code generation [13].

5.7 Malware Generators

Hidden Markov Model (HMM) based detection of metamorphic malware has been very effective in recent years. Metamorphic malwares that are created by metamorphic engines have been easily detected using HMM. To evade this, malware generators have been developed which exploit any weakness present in HMM based detection technique [27].

The proposed statistic based malware detection technique was applied to one such malware generator designed to evade HMM based detection. The results are as follows:

Testcase	% detected	# of variants detected	Total # of variants in dataset
Function30	73.5	139	190
Function5	35.2	67	190

Table 11: Results obtained for Malware Generator dataset

The above results implies that statistics based detection cannot be evaded by malware generators written specifically to evade HMM based detection. However, in order to achieve more accurate detection, a new set of statistics are to be discovered.

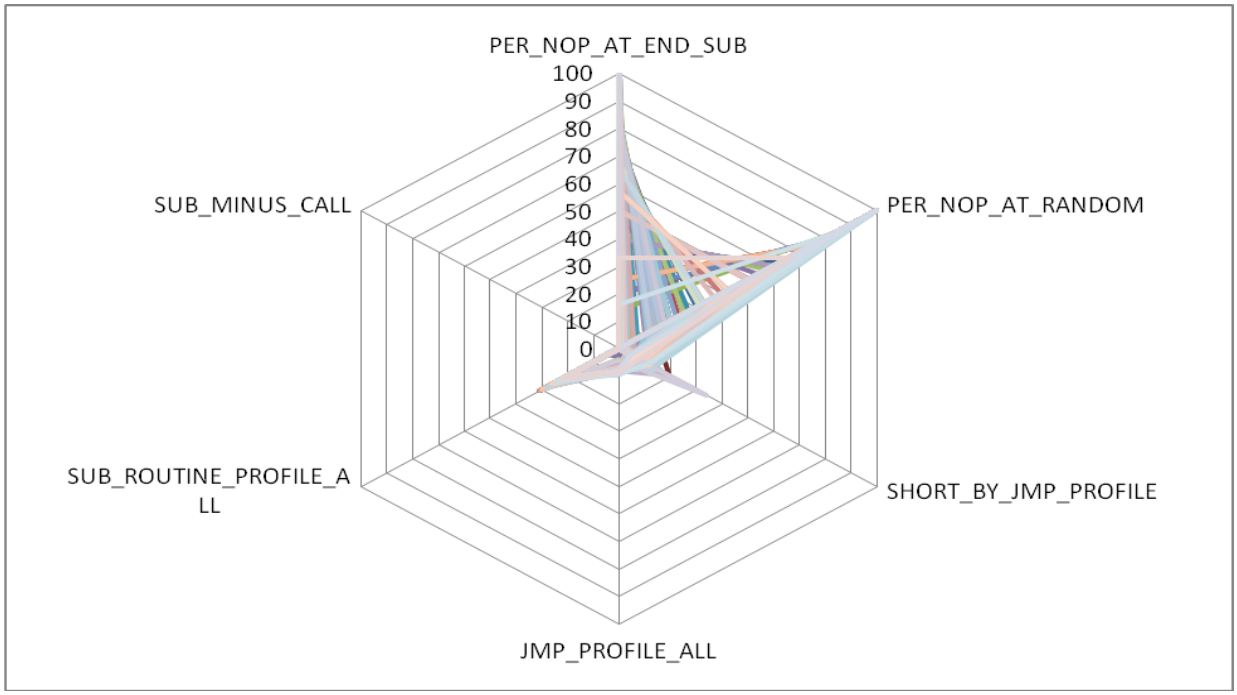

Figure 21: function 30 test data results and topology

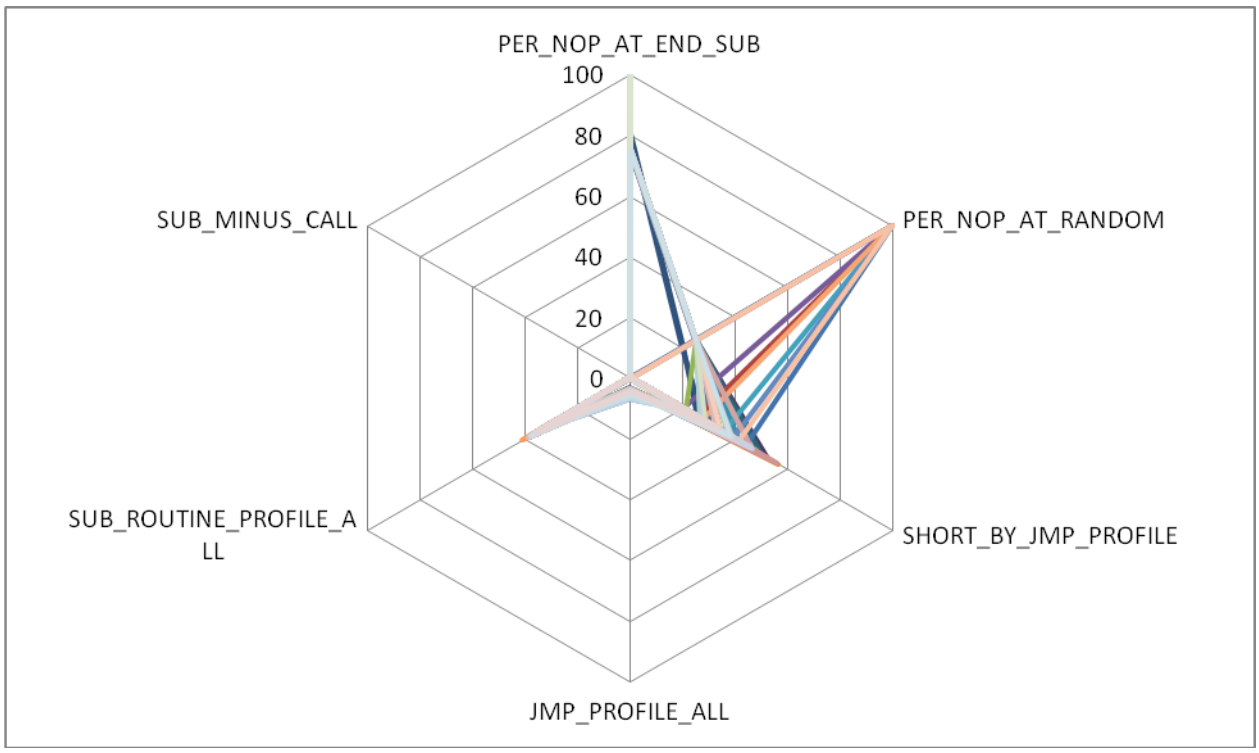

Figure 22: function 5 test data results and topology

5.8 Cygwin DLL

NGVCK executables are considered to have similar functionality as Cygwin DLL. There are 41 different Cygwin files considered for the test data[28]. Experiment aims to find out if the proposed statistics based technique can differentiate between NGVCK and Cygwin executables.

The decision tree for NGVCK malware based on statistical detection was obtained as:

```
SUB_MINUS_CALL > 0.055 (normal)
SUB_MINUS_CALL <= 0.055
        SHORT_BY_JMP_PROFILE <= 9.333 (malware)
        SHORT_BY_JMP_PROFILE > 9.333 (normal)
```

As seen in the raw data presented below (Table 12) for all 41 Cygwin files SUB_MINUS_CALL has value greater than 0.055. Therefore Cygwin executables are detected as “normal” and not as NGVCK.

PER_NOP_AT_END_SUB	PER_NOP_AT_RANDOM	SHORT_BY_JMP_PROFILE	JMP_PROFILE_ALL	SUB_ROUTINE_PROFILE_ALL	SUB_MINUS_CALL
100	0	6.557	4.108	30	0.19
94	5	10.619	6.832	27	0.124
100	0	21.525	8.17	25	0.131
56	43	12.668	8.062	23	0.111

56	43	12.668	8.044	23	0.113
100	0	20.329	8.635	25	0.128
79	20	12.5	9.004	27	0.127
100	0	24.395	8.496	25	0.134
92	7	12.387	8.529	24	0.106
90	9	16.216	8.37	26	0.136
69	30	13.876	8.248	25	0.123
100	0	23.26	7.991	23	0.116
100	0	13.953	7.517	30	0.154
100	0	20.539	8.81	25	0.124
55	44	12.959	8.717	24	0.114
100	0	23.819	7.903	23	0.12
100	0	22.474	7.94	23	0.124
54	45	13.118	8.63	24	0.114
53	46	12.452	8.597	24	0.113
53	46	12.334	8.632	24	0.112
53	46	12.476	8.614	24	0.113
53	46	12.287	8.644	24	0.111
68	31	12.879	8.581	24	0.111
90	10	14.925	5.801	23	0.116
75	24	18.962	6.665	18	0.067
56	43	12.616	8.719	24	0.113
54	45	12.195	8.653	24	0.111
53	46	13.321	8.798	24	0.111
56	43	12.684	8.728	24	0.112
54	45	13.869	8.803	24	0.112
57	42	12.939	8.648	24	0.112
76	23	18.708	6.677	17	0.064
69	30	12.366	8.75	24	0.115
94	5	15.683	8.61	25	0.117
100	0	22.826	5.008	26	0.144
54	45	14.338	8.62	24	0.112
100	0	21.446	8.879	26	0.137
100	0	20.957	9.068	26	0.13
100	0	21.247	9.112	26	0.141
100	0	20.69	8.908	26	0.137
74	25	13.947	9.205	24	0.114

Table 12: Cygwin DLL's statistic values

Next figure shows footprint of Cygwin DLLs test data.

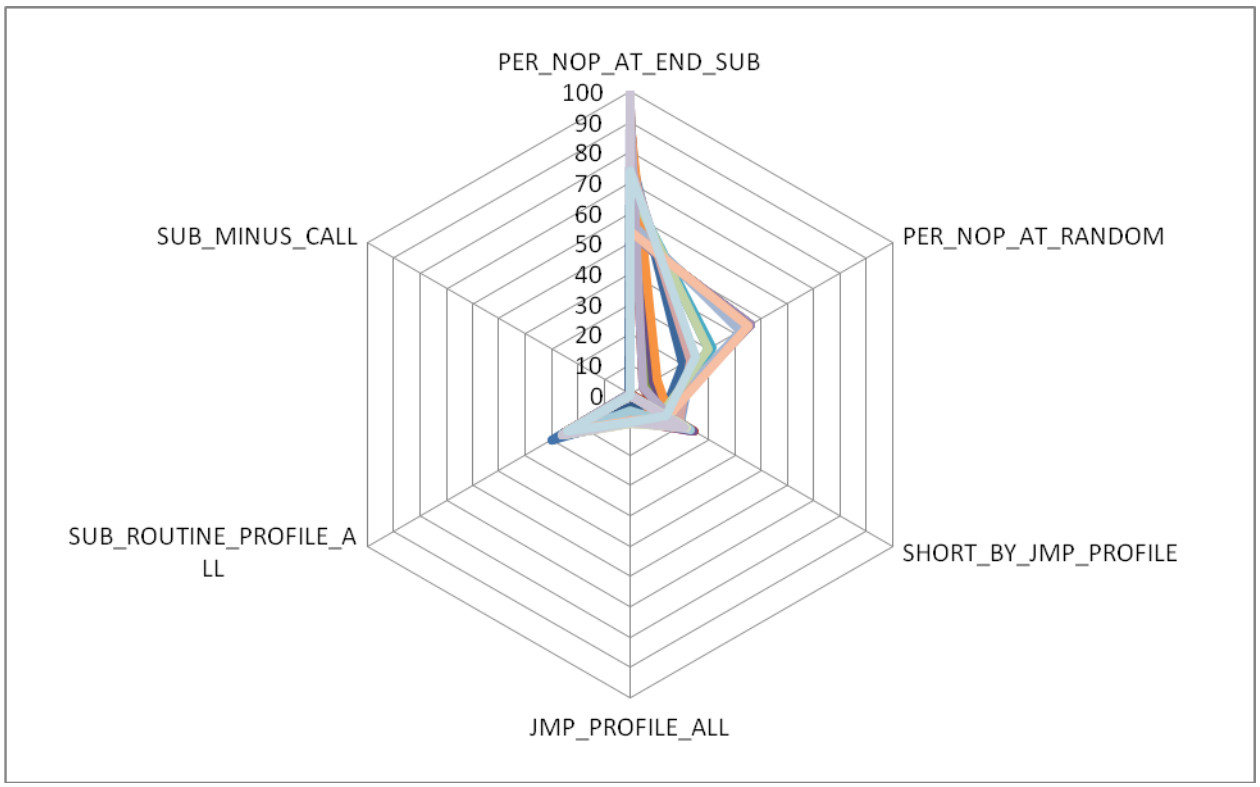

Figure 23: Footprint of Cygwin DLL test data

5.9 Comparison with other antivirus softwares.

The proposed methodology was compared against leading anti-virus software available in the market. It was found that statistics based detection was comparable with leading antivirus softwares namely Symantec and Mcafee. However, it was found that the proposed statistics based detection was better than most other antivirus softwares available in the market.

Antivirus	NGVCK	G2	MPCGEN	VCL
Avast	N	Y	Y	X
AVG	N	Y	Y	Y
Avira	N	Y	Y	N
CA Antivirus	N	Y	N	N
Proposed Methodology	**Y**	**Y**	**Y**	**Y**
F-prot	N	Y	Y	N
Symantec	**N**	**Y**	**Y**	**Y**
McAfee	**N**	**Y**	**Y**	**Y**

Table 13: Comparison against antivirus products in market

It could be observed from the table that Symantec, and McAfee do not detect NGVCK. Both the softwares claim that they have solution for NGVCK.

5.10 Effect on varying the number of statistics.

In order to optimize and minimze the number of statistics that are captured, different testcases were conducted to check the error rate achieved by deleting some statistics. All the 6 statistics were divided into 3 categories namely Garbage instruction detection statistics (PER_NOP_AT_END_SUB, PER_NOP_AT_RANDOM), Code reordering detection statistics (JMP_PROFILE_ALL and SHORT_BY_JMP_PROFILE) and subroutine permutation detection statistics (SUB_ROUTINE_PROFILE_ALL and SUB_MINUS_CALL).

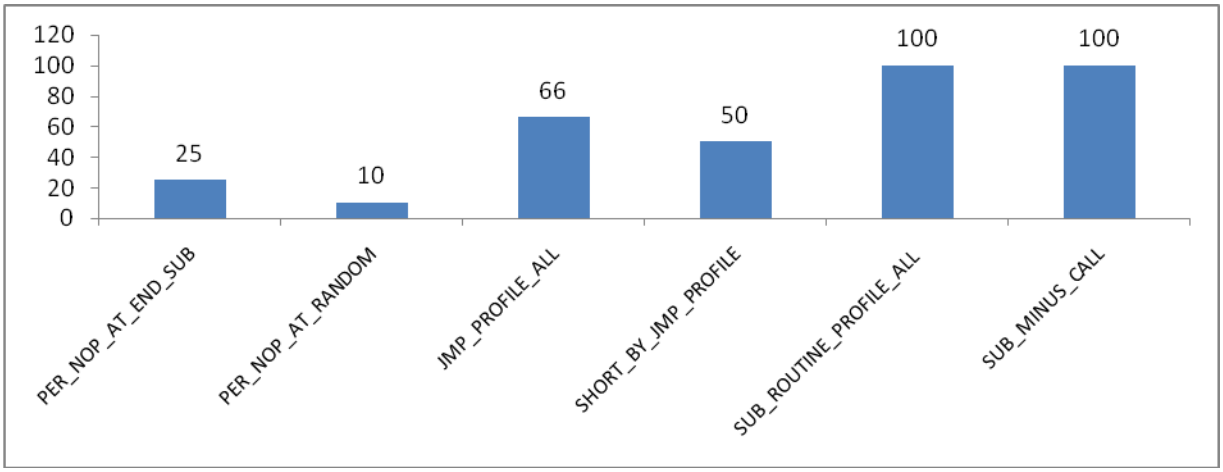

Figure 24: Percentage of false positives on removing each category of statistics

6.0 Conclusion

In this project, a new technique has been proposed for detecting metamorphic malwares based on statistics. It has been found that statistics based detection can be an effective tool in detecting metamorphic malwares.

All the four test malware variants such as NGVCK, VCL, G2 and MPCGEN family were detected using the proposed technique. A decision tree obtained on the basis of statistics, can act as a signature to a type of malware variants. C4.5 has been used as the classifier for generating decision trees based on test data, thereby proving the versatility of C4.5.

The six statistics chosen (PER_NOP_AT_END_SUB, PER_NOP_AT_RANDOM, JMP_PROFILE_ALL, SHORT_BY_JMP_PROFILE, SUB_ROUTINE_PROFILE_ALL and SUB_MINUS_CALL) counteract specific types of code obfuscation namely garbage code insertion, sub-routine permutation and code-reordering. If a malware uses a different type of code obfuscation technique, then appropriate statistics need to be developed to counteract those techniques, so as to detect them.

McAfee, Symantec are observed to be identifying all the malwares except NGVCK. McAfee and Symantec claim that they have solution for NGVCK but results differ other way. McAfee and Symantec are more comprehensive softwares. Proposed technique has edge with respect to these two in the size of executables, and system resources required.

Statistics based detection is based on static analysis of assembly code instructions. It is light weight and easy to implement. The accuracy of the system depends on the quality of dis-assembler. The higher the quality of dis-assembler, the accurate the whole system would be.

7.0 Future Work

The accuracy of the proposed system depends on the statistics used. The statistics should be chosen in such as way so that they are mutually exclusive to each other. It would be interesting to develop a statistic generator module which helps in automatically identifying new statistics.

Malware writers use various techniques which makes it hard to disassemble an exe. Therefore, considerable research is required for developing better dis-assemblers.

Clustering of exe's based on the obtained statistics would help in developing threshold values for each family of malware. It would be interesting to see the effect of using weighted statistics, on the accuracy of malware detection.

The speed of malware detection using this technique needs to be investigated. With the help of proper heuristics, the speed of detection could be vastly improved. The accuracy of malware detection on combining statistics based malware detection with other existing techniques needs to be looked into. A good combination would be one with Anomaly based detection as it would give a chance to detect new malwares, as our proposed methodology will handle the variants of known malwares.

APPENDICES

Appendix A. NGVCK Testdata results

JMP_PROFILE_ALL	SUB_ROUTINE_PROFILE_ALL	SUB_MINUS_CALL
5.993	11	0.004
6.349	12	0.012
7.184	13	0.01
7.573	13	0.012
6.695	12	0.006
7.495	13	0.006
6.715	12	0.015
5.95	13	0.021
6.589	13	0.019
5.847	11	0.01
7.753	12	0.002
6.144	12	0.008
5.361	12	0.016
7.407	13	0.006
6.193	12	0.016
5.882	12	0.018
6.857	13	0.015
6.275	12	0.012
5.986	12	0.018
6.222	12	0.011
5.932	12	0.015
6.744	12	0.01
5.242	12	0.02
6.107	12	0.017
6.653	13	0.012
6.776	13	0.018
6.167	12	0.011
7.018	13	0.007
7.917	13	0
7.677	13	0.004
5.917	12	0.012
6.883	13	0.013
5.945	12	0.011
5.97	12	0.011
6.731	13	0.019
6.371	12	0.012
7.157	13	0.018
7.171	12	0.008

6.859	13	0.013
6.364	12	0.015
6.119	13	0.021
6.479	12	0.011
5.812	12	0.014
6.54	13	0.015
6.331	12	0.011
7.392	14	0.016
7.632	13	0.01
6.77	12	0.006
6.237	13	0.016
6.366	12	0.008
6.465	12	0.012
6.939	12	0.008
5.609	12	0.015
5.947	12	0.011
7.216	13	0.008
6.549	13	0.016
6.604	12	0.011
6.992	13	0.011
6.584	13	0.014
6.993	12	0.012
5.577	12	0.015
5.95	12	0.012
6.472	12	0.004
7.054	13	0.012
5.527	11	0.01
6.723	13	0.013
7.555	12	-0.002
6.838	12	0.004
7.356	12	0.004
7.283	12	0.004
5.472	12	0.019
5.684	12	0.017
6.703	13	0.014
8.125	13	0.004
7.602	13	0.006
5.929	12	0.016
5.684	12	0.017
6.151	12	0.016
6.794	13	0.017
6.043	12	0.016
6.055	13	0.022

7.113	12	0.006
4.924	12	0.025
5.943	12	0.014
7.042	13	0.016
6.522	12	0.008
5.513	12	0.017
6.25	12	0.011
5.273	12	0.02
5.985	12	0.019
6.183	12	0.009
6.426	12	0.01
6.186	12	0.012

Appendix B. VCL Testdata results

JMP_PROFILE_ALL	SUB_ROUTINE_PROFILE_ALL	SUB_MINUS_CALL
5.183	11	-0.043
15.425	16	-0.085
6.498	12	-0.043
9.063	12	-0.041

Appendix C. G2 Testdata results

SHORT_BY_JMP_PROFILE	JMP_PROFILE_ALL	SUB_ROUTINE_PROFILE_ALL	SUB_MINUS_CALL
0	6.742	13	0.056
7.143	5.469	13	0.062
7.143	5.882	13	0.063

Appendix D. MPCGEN Testdata results

SHORT_BY_JMP_PROFILE	JMP_PROFILE_ALL	SUB_ROUTINE_PROFILE_ALL	SUB_MINUS_CALL
9.091	6.509	15	0.065
7.143	5.833	15	0.088
8.333	9.756	15	0.033
0	7.303	14	0.051
0	5.164	15	0.085
6.25	5.556	14	0.08
17.647	6.071	16	0.075
8.333	8.108	14	0.041
6.25	6.426	14	0.072
0	5.34	15	0.078
17.647	6.967	15	0.061
8.333	8.108	14	0.041

8.333	6.091	15	0.066
18.75	7.805	16	0.059

Appendix E. Source Code

```
/******************  AssemblyParser.java  *************************/
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;


public class AssemblyParser
{

	HashMap<String, Integer> instructionSet = new HashMap<String, Integer>();
	Integer followedByNOPOrEndSub = 0;
	Integer followedByOtherIns = 0;
	public AssemblyParser()
	{
		//Initialize the 8086 instructions into a hashmap
		initialize8086InstructionSet();
	}


	private void initialize8086InstructionSet()
	{
		//Reads 8086-config.ini and initializes each 8086 instruction
	    File file = new File("8086-config.ini");
	    StringBuffer contents = new StringBuffer();
	    BufferedReader reader = null;

	    try
	    {
	        reader = new BufferedReader(new FileReader(file));
	        String text = null;

	        // repeat until all lines is read
	        while ((text = reader.readLine()) != null)
	        {
	              instructionSet.put(text, 0);
	       }
	    }
	    catch (Exception e)
	    {
	        e.printStackTrace();
	    }
	    finally
```

```
        {
            try
            {
                if (reader != null)
                {
                    reader.close();
                }
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }


    }


    public void extractPrimaryStats(String pathToFile)
    {
          //The main method which reads an input file and generates statistics out
of it.
          try
          {
              BufferedReader in = new BufferedReader(new FileReader(pathToFile));
              String eachLineAsStr;
              Boolean NOPFlagRaised = false;
              while ((eachLineAsStr = in.readLine()) != null)
              {
                 //String[] tokenArr1 = eachLineAsStr.split(" ");
                 //String[] tokenArr = eachLineAsStr.split("\\t");
                 //System.out.println(eachLineAsStr);
                 String[] tokenArr = eachLineAsStr.split(" ");

                 /*
                 Matcher m;
                 m = Pattern.compile(".*ADD
.*,[^(\\d*D)(.*H)(\\d*B)(\\w*^\\w)]").matcher(eachLineAsStr.toUpperCase());
                 if (m.find())
                 {
                        System.out.println("FOUND:" + eachLineAsStr);
                 }
                 */
                 Matcher m;
                 m =
Pattern.compile(".*\\sSHORT\\s.*").matcher(eachLineAsStr.toUpperCase());
                 if (m.find())
                 {
                        //instructionSet.put("SHORT",
instructionSet.get("SHORT")+1);
                 }
                 m = null;
                 //Pattern matching for loc_XXXXX or sub_XXXXX:

                 {


                        m =
Pattern.compile("LOC_.*").matcher(eachLineAsStr.toUpperCase());
                        if (m.find() &&
eachLineAsStr.trim().toUpperCase().startsWith("LOC"))
                        {
                               instructionSet.put("LOC_ROUTINE",
```

```
instructionSet.get("LOC_ROUTINE")+1);
                        }

                        m = null;
                        m =
Pattern.compile("SUB_.*:").matcher(eachLineAsStr.trim().toUpperCase());
                        if (m.find() &&
eachLineAsStr.trim().toUpperCase().startsWith("SUB"))
                        {
                              instructionSet.put("SUB_ROUTINE",
instructionSet.get("SUB_ROUTINE")+1);
                        }
                        else
                        {
                              m = null;
                              m =
Pattern.compile(".*:").matcher(eachLineAsStr.trim().toUpperCase());
                              if (m.find())
                              {
                                    instructionSet.put("SUB_ROUTINE",
instructionSet.get("SUB_ROUTINE")+1);
                              }
                        }
                  }



                   for (int i = 0; i < tokenArr.length; i++)
                   {

                        //System.out.println(tokenArr[i].trim());
                        if ((tokenArr[i].trim().length() > 1))
                        {
                              if
(isAnInstruction(tokenArr[i].trim().toUpperCase()))
                              {

      //instructionSet.put(tokenArr[i].trim().toUpperCase(),
instructionSet.get(tokenArr[i].trim().toUpperCase())+1);
                                    if (NOPFlagRaised)
                                    {

      //System.out.println(tokenArr[i].trim());
                                          if
(tokenArr[i].trim().toUpperCase().equalsIgnoreCase("NOP"))
                                          {
                                                //System.out.println("other43");
                                                followedByNOPOrEndSub++;
                                          }
                                          else
                                          {
                                                //System.out.println("other");
                                                followedByOtherIns++;
                                          }
                                          NOPFlagRaised = false;
                                    }
                                    if
(tokenArr[i].trim().toUpperCase().equalsIgnoreCase("NOP"))
                                    {
                                          //System.out.println("found " +
eachLineAsStr);
                                          NOPFlagRaised = true;
                                          i = tokenArr.length+1;
```

```
                                        }
                                }
                                else
                                {
                                        if (NOPFlagRaised)
                                        {

        //System.out.println("here2"+tokenArr[i].trim());
                                                followedByNOPOrEndSub++;
                                                NOPFlagRaised = false;
                                        }

                                }
                        }
                        else
                        {
                                if (NOPFlagRaised)
                                {
                                        //System.out.println("here5");
                                        followedByNOPOrEndSub++;
                                        NOPFlagRaised = false;
                                }

                        }
                  }

                if (tokenArr.length == 0)
                {
                        if (NOPFlagRaised)
                        {
                                //System.out.println("here7");
                                followedByNOPOrEndSub++;
                                NOPFlagRaised = false;
                        }
                }
            }
            in.close();
        }
        catch (IOException e)
        {
                e.printStackTrace();
        }
    }

    private boolean isAnInstruction(String testStr) {
        // Checks to see if the given token is an instruction or not.

        String[] tokenArr = testStr.split("\\t");
        Boolean isFoundFlag = false;
        if (tokenArr.length > 0)
        {
                for (int k=0; k<tokenArr.length; k++)
                {
                        if (instructionSet.containsKey(tokenArr[k]))
                        {
                                instructionSet.put(tokenArr[k].trim().toUpperCase(),
instructionSet.get(tokenArr[k].trim().toUpperCase())+1);
                                k = tokenArr.length+1;
                                isFoundFlag = true;
                        }
                }
        }
        return isFoundFlag;
```

```
}


public int returnStatsFor(String ins)
{
	// Returns the statistics count for a particular instruction
	return instructionSet.get(ins.toUpperCase());
}

public double returnJumpProfile()
{
	// Returns the statistics of all the JMP variants combined.
	double jmpInsCounter = 0.0;
	jmpInsCounter +=
	instructionSet.get("JA")+
	instructionSet.get("JAE")+
	instructionSet.get("JB")+
	instructionSet.get("JBE")+
	instructionSet.get("JC")+
	instructionSet.get("JCXZ")+
	instructionSet.get("JE")+
	instructionSet.get("JG")+
	instructionSet.get("JGE")+
	instructionSet.get("JL")+
	instructionSet.get("JLE")+
	instructionSet.get("JNA")+
	instructionSet.get("JNAE")+
	instructionSet.get("JNB")+
	instructionSet.get("JNBE")+
	instructionSet.get("JNC")+
	instructionSet.get("JNE")+
	instructionSet.get("JNG")+
	instructionSet.get("JNGE")+
	instructionSet.get("JNL")+
	instructionSet.get("JNLE")+
	instructionSet.get("JNO")+
	instructionSet.get("JNP")+
	instructionSet.get("JNS")+
	instructionSet.get("JO")+
	instructionSet.get("JP")+
	instructionSet.get("JPE")+
	instructionSet.get("JPO")+
	instructionSet.get("JS")+
	instructionSet.get("JZ")+
	instructionSet.get("JCXZ")+
	instructionSet.get("JECXZ")+
	instructionSet.get("JMP");

	return jmpInsCounter;
}

public Integer getFollowedByNOPOrEndSub() {
	return followedByNOPOrEndSub;
}


public void setFollowedByNOPOrEndSub(Integer followedByNOPOrEndSub) {
	this.followedByNOPOrEndSub = followedByNOPOrEndSub;
}


public Integer getFollowedByOtherIns() {
	return followedByOtherIns;
```

```
	}


	public void setFollowedByOtherIns(Integer followedByOtherIns) {
		this.followedByOtherIns = followedByOtherIns;
	}


	public int returnNumASMLoc()
	{
		// Returns the total number of Assembly lines of code
		int numASMLoc = 0;
	    Set entries = instructionSet.entrySet();
	    Iterator it = entries.iterator();
	    while (it.hasNext())
	    {
	      Map.Entry<String, Integer> entry = (Map.Entry) it.next();
	     // if (entry.getValue() > 0)
	      {
	         numASMLoc += entry.getValue();
	      }
	    }

	    return numASMLoc;
	}
	public void printAllStats(int mode)
	{
		//Method to print all statistics

	    // List the entries using entrySet()
	    Set entries = instructionSet.entrySet();
	    Iterator it = entries.iterator();
	    while (it.hasNext())
	    {
	      Map.Entry<String, Integer> entry = (Map.Entry) it.next();
	     // if (entry.getValue() > 0)
	      {
	         if (mode == 0)
	         {
	               System.out.println(entry.getKey());
	         }
	         if (mode == 1)
	         {
	               System.out.println(entry.getValue());
	         }
	         if (mode == 2)
	         {
	               System.out.println(entry.getKey() + "," + entry.getValue());
	         }

	      }
	    }
	}

}
/***************** ASMFileFilter.java **********************/

import java.io.*;

public class ASMFileFilter implements FilenameFilter
{
	public boolean accept(File dir, String name)
	{
```

```
		// Returns a list of files in the current directory that end with .asm
extension.
		return name.endsWith(".asm");
	}
}
/***************** ExeFileFilter.java **********************/
import java.io.*;

public class ExeFileFilter implements FilenameFilter
{
	public boolean accept(File dir, String name)
	{
		//Return all files in the current directory that end with .EXE extension.
		return name.endsWith(".exe");
	}
}
/***************** StatisticsManager.java **********************/

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.text.DecimalFormat;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Scanner;


public class StatisticsManager
{

	/**
	 * @param args
	 */
	public static void main(String[] args)
	{
		// Statistics Manager is the umbrella class that takes care of generating
the statistics
		// from EXE's

		Scanner scanner = new Scanner(System.in);
		System.out.print("Run Test [malware/normal] : ");
		String testType = scanner.nextLine();
		System.out.print("Path : ");
		File rootASMDir = null;
		rootASMDir = new File(scanner.nextLine());
		System.out.print("File name Suffix : ");
		String fileSuffix = scanner.nextLine();
		System.out.print("Extract ASM from EXE's [Y/N] : ");
		Boolean shouldExtractEXE = false;
		shouldExtractEXE = (scanner.nextLine().trim().equalsIgnoreCase("Y")) ?
true : false;
		System.out.print("Generate Multiple versions from NGVCK [Y/N] : ");
		Boolean shouldGenMulNGVCK = false;
		shouldGenMulNGVCK = (scanner.nextLine().trim().equalsIgnoreCase("Y")) ?
true : false;

		//ExeFileFilter onlyEXE = new ExeFileFilter();
		//UtilityManager.print(onlyEXEArr);

		// uncomment to extract asm code from all exe's in computer
		if (shouldExtractEXE)
		{
```

```
                    File rootDir = new File("C:\\Program Files (x86)");
                    File[] onlyEXEArr = SystemInfoManager.listFilesAsArray(rootDir,
new ExeFileFilter(), true);

                    for (int i = 20; i < onlyEXEArr.length; i++)
                    {

      UtilityManager.extractASMCodeFromEXE(onlyEXEArr[i].getAbsolutePath(),i);

                    }
             }
             //

             //code to generate multiple versions of malware from NGVCK
             if (shouldGenMulNGVCK)
             {
                    for (int i = 1; i < 10; i++)
                    {
                           UtilityManager.extractASMCodeFromNGVCK(i);
                    }
             }
             //


             // code to extract statistics from ASM files
             /*
             if (testType.trim().equalsIgnoreCase("malware"))
             {
                    //rootASMDir = new File("C:\\Documents and Settings\\admin\\My
Documents\\Downloads\\KITS\\ngvck030s");
                    rootASMDir = new File("C:\\Documents and Settings\\admin\\My
Documents\\Downloads\\KITS\\ngvck030s\\ngvck-test");
             }
             else
             {
                    rootASMDir = new File("C:\\Documents and Settings\\admin\\My
Documents\\Downloads\\AssemblyParser\\normal exe");
             }
             */

             if (!shouldExtractEXE)
             {
             File[] onlyASMArr = SystemInfoManager.listFilesAsArray(rootASMDir, new
ASMFileFilter(), true);


      System.out.print("PER_NOP_AT_END_SUB,PER_NOP_AT_RANDOM,STATS_NOP_PER,STATS_XCHG
_PER,STATS_JMP_PER,SHORT_BY_JMP_PROFILE,JMP_PROFILE_ALL,SUB_ROUTINE_PROFILE_ALL,SUB_MI
NUS_CALL");
       // Use an Iterator to traverse the mappings in the TreeMap.

       /*
        AssemblyParser asmParserObj1 = new AssemblyParser();
            Iterator iterator = asmParserObj1.instructionSet.entrySet().iterator();

       while (iterator.hasNext()) {
           Map.Entry entry = (Map.Entry) iterator.next();
           System.out.print(entry.getKey() + ",");
       }


       generateTemplateFile();
       */
```

```
		System.out.println();
		for (int i = 0; i < onlyASMArr.length; i++)
		{
			AssemblyParser asmParserObj = new AssemblyParser();
			asmParserObj.extractPrimaryStats(onlyASMArr[i].getAbsolutePath());
			int numASMLOC = 0;
			String statOut = "";
			//System.out.print(i+1 + ",");
			statOut += "";//(i+1) + ","
			//statOut +=
onlyASMArr[i].getName() + ",";
                                                                // +
Math.round(onlyASMArr[i].length()/1024)
                                                                 //+ "KB"
                                                                // + ","
                                                                // +
asmParserObj.returnNumASMLoc()
                                                                // + ","
			if ((asmParserObj.returnStatsFor("NOP")) == 0)
			{
				statOut += "0,0";
			}
			else
			{
				statOut +=
((asmParserObj.getFollowedByNOPOrEndSub()*100)/(asmParserObj.returnStatsFor("NOP")) );
				statOut      += ",";
				statOut +=
((asmParserObj.getFollowedByOtherIns()*100)/(asmParserObj.returnStatsFor("NOP")) );
			}
			if ((asmParserObj.returnNumASMLoc()) == 0)
			{
				statOut += ",0,0,0,0";
			}
			else
			{
				double tempStore = 0.0;
				DecimalFormat df = new DecimalFormat("#.###");

				tempStore =
((asmParserObj.returnStatsFor("NOP")*100)/asmParserObj.returnNumASMLoc());
				statOut      += ",";
				statOut +=   df.format(tempStore);
				statOut      +=     ",";
				tempStore =
((double)((asmParserObj.returnStatsFor("XCHG")*100)/(asmParserObj.returnNumASMLoc())))
;
				statOut +=   df.format(tempStore);
				statOut      += ",";
				tempStore =
((double)((asmParserObj.returnStatsFor("JMP")*100)/(asmParserObj.returnNumASMLoc())));
				statOut +=   df.format(tempStore);
				statOut      += ",";
				tempStore =
((asmParserObj.returnStatsFor("SHORT")*100)/(asmParserObj.returnJumpProfile()));
				statOut +=   df.format(tempStore);
				statOut      += ",";
				tempStore =
((asmParserObj.returnJumpProfile()*100)/(asmParserObj.returnNumASMLoc()));
				statOut +=   df.format(tempStore);
				statOut      += ",";
				tempStore = (((asmParserObj.returnStatsFor("LOC_ROUTINE") +
asmParserObj.returnStatsFor("SUB_ROUTINE"))*100)/(asmParserObj.returnNumASMLoc()));
```

```
                        statOut +=    df.format(tempStore);
                        statOut       += ",";
                        tempStore =
(double)((((asmParserObj.returnStatsFor("SUB_ROUTINE")*1)+(asmParserObj.returnStatsFor
("LOC_ROUTINE")*1)-(asmParserObj.returnStatsFor("CALL")*1)-
(asmParserObj.returnJumpProfile()))/asmParserObj.returnNumASMLoc()));
                        statOut +=    df.format(tempStore);

                    }
                    /*
              Iterator iteratorInside =
asmParserObj.instructionSet.entrySet().iterator();

              while (iteratorInside.hasNext()) {
                  Map.Entry entry = (Map.Entry) iteratorInside.next();
                  statOut += entry.getValue() + ",";
              }
              */
              System.out.println(statOut + "," + testType);

              try
              {
                  // Create file
                  FileWriter fstream = new FileWriter("results_" + testType + "_" +
fileSuffix + ".csv",true);
                  BufferedWriter out = new BufferedWriter(fstream);
                  out.write(statOut + "," + testType);
                  out.write("\r\n");
                  //Close the  output stream
                  out.close();
              }
              catch (Exception e)
              {//Catch exception if any
                    System.err.println("Error: " + e.getMessage());
              }

            }

            }
      }

      private static void generateTemplateFile() {
            // This method  generates the template file that can be used with C4.5
program
       AssemblyParser asmParserObj1 = new AssemblyParser();
       Iterator iteratorInside = asmParserObj1.instructionSet.entrySet().iterator();
       try
       {
           // Create file
           FileWriter fstream = new FileWriter("template.txt",true);
           BufferedWriter out = new BufferedWriter(fstream);
           while (iteratorInside.hasNext())
           {
               Map.Entry entry = (Map.Entry) iteratorInside.next();
               out.write(entry.getKey() + ": continuous.");
               out.write("\r\n");
           }

           //Close the  output stream
           out.close();
       }
       catch (Exception e)
       {//Catch exception if any
```

```
                System.err.println("Error: " + e.getMessage());
            }

        }

}
/***************** SystemInfoManager.java ***********************/
import java.io.File;
import java.io.FilenameFilter;
import java.util.Collection;
import java.util.Vector;

//code obtained from http://snippets.dzone.com/posts/show/1875

public class SystemInfoManager {

        public static File[] listFilesAsArray(
                        File directory,
                        FilenameFilter filter,
                        boolean recurse)
        {
                Collection<File> files = listFiles(directory,
                                filter, recurse);
        //Java4: Collection files = listFiles(directory, filter, recurse);

                File[] arr = new File[files.size()];
                return files.toArray(arr);
        }

        public static Collection<File> listFiles(
        // Java4: public static Collection listFiles(
                        File directory,
                        FilenameFilter filter,
                        boolean recurse)
        {
                // List of files / directories
                Vector<File> files = new Vector<File>();
        // Java4: Vector files = new Vector();

                // Get files / directories in the directory
                File[] entries = directory.listFiles();

                // Go over entries
                try
                {
                for (File entry : entries)
                {
        // Java4: for (int f = 0; f < files.length; f++) {
        // Java4:     File entry = (File) files[f];

                        // If there is no filter or the filter accepts the
                        // file / directory, add it to the list
                        if (filter == null || filter.accept(directory, entry.getName()))
                        {
                                files.add(entry);
                        }

                        // If the file is a directory and the recurse flag
                        // is set, recurse into the directory
                        if (recurse && entry.isDirectory() &&
!entry.getAbsolutePath().contains("IDA Free"))
                        {
                                files.addAll(listFiles(entry, filter, recurse));
```

```
                        }
                }
                }
                catch(NullPointerException e)
                {

                }
                // Return collection of files
                return files;
        }
}

/***************** TXTFileFilter.java **********************/

import java.io.*;

public class TXTFileFilter implements FilenameFilter
{
        public boolean accept(File dir, String name)
        {
                // Returns all files in the current directory that end with .txt
extension
                return name.endsWith(".txt");
        }
}

/***************** UtilityManager.java **********************/

import java.awt.AWTException;
import java.awt.Robot;
import java.awt.event.KeyEvent;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

// Class which takes care of running external programs such as IDA Pro.
public class UtilityManager
{
        // Prints the path of current file being processed.
        public static void print(File[] result)
        {
                for (int i = 0; i < result.length; i++)
                {
                        System.out.println(i + ". " + result[i].getAbsolutePath());
                }
        }

        //Auto script that extracts asm code from exe by running IDA Pro command line
options.
        public static synchronized Process
         extractASMCodeFromEXE(String filename, int index)
        {
                        String cmd =  "C:\\Program Files (x86)\\IDA Free\\idag.exe" + " -A
-Sanalysis.idc " + '"' + filename + '"';
/*
                        FileOutputStream fos;
                            DataOutputStream dos;
```

```
                        try {

                          File file= new File(index+".bat");
                          fos = new FileOutputStream(file);
                          dos=new DataOutputStream(fos);
                          dos.writeChars(cmd);

                        } catch (IOException e) {
                          e.printStackTrace();

                        }
*/
                    Runtime run = Runtime.getRuntime();
                    Process pr = null;
                    try
                    {
                           pr = run.exec(cmd);
                           System.out.println("START: " + filename);
                           System.out.println(cmd);
                           try
                           {

                               Robot robot = new Robot();
                               // Simulate a key press
                               robot.delay(2000);
                               robot.keyPress(KeyEvent.VK_ENTER);
                               robot.keyPress(KeyEvent.VK_SPACE);
                           }
                           catch (AWTException e)
                           {
                               e.printStackTrace();
                           }
                           pr.waitFor();
                           while (checkIsProcessRunning())
                           {
                                  Thread.sleep(5000);
                           }
                           System.out.println("END: " + filename);
                    }
                    catch (Exception e)
                    {
                           // TODO Auto-generated catch block
                           e.printStackTrace();
                    }
                    return pr;

             }

       // Auto Script to different types of NGVCK virus
       public static synchronized Process
        extractASMCodeFromNGVCK(int index)
       {
                    String cmd =  "C:\\Documents and Settings\\admin\\My
Documents\\Downloads\\KITS\\ngvck030s\\NGVCK.exe";
                    Runtime run = Runtime.getRuntime();
                    Process pr = null;
                    try
                    {
                           pr = run.exec(cmd);
                           System.out.println("START: " + index);
                           System.out.println(cmd);
                           try
```

```
                {

                    Robot robot = new Robot();
                    // Simulate a key press
                    robot.delay(7000);
                    robot.keyPress(KeyEvent.VK_ENTER);
                    robot.keyPress(KeyEvent.VK_TAB);
                    robot.keyPress(KeyEvent.VK_TAB);
                    robot.keyPress(KeyEvent.VK_TAB);
                    robot.keyPress(KeyEvent.VK_TAB);
                    robot.keyPress(KeyEvent.VK_END);
                    if (index < 10)
                    {
                       robot.keyPress(48+index);
                    }
                    else
                    {
                       if (index < 100)
                       {
                              robot.keyPress(((int)index/10) + 48);
                              robot.keyPress(((int)index%10) + 48);
                       }
                       else
                       {
                              robot.keyPress(((int)index/100) + 48);
                              robot.keyPress(((int)(index-100)/10) + 48);
                              robot.keyPress(((int)(index-100)%10) + 48);

                       }
                    }
                    robot.keyPress(KeyEvent.VK_TAB);
                    robot.keyPress(KeyEvent.VK_ENTER);
                    robot.keyPress(KeyEvent.VK_ENTER);
                    robot.keyPress(KeyEvent.VK_SPACE);
                    robot.keyPress(KeyEvent.VK_TAB);
                    robot.keyPress(KeyEvent.VK_ENTER);
                    robot.keyPress(KeyEvent.VK_ENTER);
                    robot.delay(2000);
                    robot.keyPress(KeyEvent.VK_ENTER
                             );
                }
                catch (AWTException e)
                {
                    e.printStackTrace();
                }
                pr.waitFor();
                while (checkIsProcessRunning())
                {
                       Thread.sleep(5000);
                }
                System.out.println("END: " + index);
            }
            catch (Exception e)
            {
                   // TODO Auto-generated catch block
                   e.printStackTrace();
            }
            return pr;

      }

// Check to see if IDA Pro is still running.
public static Boolean checkIsProcessRunning()
```

```
	{
		Runtime runtime = Runtime.getRuntime();
		 String cmds[] = {"cmd", "/c", "tasklist"};
		 Process proc;
		 Boolean isRunning = false;
		   try
		   {
			  proc = runtime.exec(cmds);

		 InputStream inputstream = proc.getInputStream();
		 InputStreamReader inputstreamreader = new
InputStreamReader(inputstream);
		 BufferedReader bufferedreader = new
BufferedReader(inputstreamreader);
		 String line;

		 while ((line = bufferedreader.readLine()) != null)
		 {
		     if (line.contains("idag.exe"))
		     {
		      isRunning = true;
		     }
		 }
		   }
		   catch (Exception e)
		   {
			  // TODO Auto-generated catch block
			  e.printStackTrace();
		   }
	return isRunning;
	}

}
```

9.0 REFERENCES

[1] Srilatha Attaluri, S. McGhee, Mark Stamp. Profile hidden Markov models and metamorphic virus detection, Journal in Computer Virology.

[2] A Venkatesan. Code obfuscation and metamorphic virus detection, Masters Thesis, San Jose State University.

[3] S. McGhee. Pairwise alignment of metamorphic computer viruses, Masters Thesis, San Jose State University.

[4] Mark Stamp, Information Security: Principles and Practice, Wiley Interscience, September 2005, ISBN: 0-471-73848-4

[5] Karnik, Abhishek; Goswami, Suchandra; Guha, Ratan: Detecting Obfuscated Viruses Using Cosine Similarity Analysis. First Asia International Conference on Modelling & Simulation, 2007, IEEE, Page(s):165 – 170

[6] M. Chouchane and Arun Lakhotia. Using Engine Signature to Detect Metamorphic Malware, Proceedings of the Fourth ACM Workshop on Recurring Malcode (WORM), November 2006, pp. 73-78

[7] Mohamed R. Chouchane, Andrew Walenstein, and Arun Lakhotia. Statistical Signatures for Fast Filtering of Instruction-substituting Metamorphic Malware, Proceedings of the 2007 ACM workshop on Recurring malcode, 2007, pp. 31 -37.

[8] P Vinod, R Jaipur, V Laxmi, MS Gaur. Survey on Malware Detection Methods, http://sites.google.com/site/advnwsec09/SurveyonMalwaredetectiontechniques.PDF

[9] S. Govindaraj. Practical detection of metamorphic computer viruses, Masters Thesis, San Jose State University.

[10] P. Desai. Towards an undetectable computer virus, Masters Thesis, San Jose State University.

[11] P. Szor, The Art of Computer Virus Research and Defense, Addison-Wesley, 2005

[12] Peter Szor and Peter Ferrie, "Hunting for Metamorphic," Virus Bulletin Conference, September 2001, pp. 123-144

[13] Peter Szor , "Zmist opportunities," Virus Bulletin Conference, March 2001, pp. 6-7

[14] Matt Webster and Grant Malcolm, Detection of metamorphic and virtualization-based malware using Algebraic Specification, 17 Annual EICAR Conference 2008

[15] S. Momina Tabish, M. Zubair Shafiq, Muddassar Farooq, Malware Detection using Statistical Analysis of Byte level content, CSI-KKD 2009

[16] S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, "Signature Generation and Detection of Malware Families," in Proceeding of the 13th Australasian Conference on Information Security and Privacy, Wollongong, Australia, July 2008, pp. 336–349.

[17] Felix Leder, Bastian Steinbock, Peter Martini, Classsification and detection of metamorphic malware using value set analysis.

[18] Mohamed R.Chouchane, Andrew Walenstein and Arun Lakhotia," Statistical Signature for Fast Filtering of Instruction-substituting Metamorphic Malware", In Proc. Worm07, November 2, 2007", Alexandria, Virgina, USA, ACM Press.

[19] Frederic Perriot, Peter Szor, and Peter Ferrie. Striking similarites: Win32/simile and metamorphic virus code. Technical report, Symantec, 2003.

[20] Yan Zhou and Meador Inge, Malware Detection using Adaptive Data Compression

[21] Myles Jordan. Dealing with metamorphism. Virus Bul letin, pages 4–6, Octomber 2002.

[22] Prabhat K. Singh and Arun Lakhotia. Analysis and detection of computer viruses and worms: an annotated bibliography. SIGPLAN Not., 37(2):29–35, 2002.

[23] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[24] E. Konstantinou, "Metamorphic virus: Analysis and detection," 2008,Technical Report RHUL-MA-2008-2, Search Security Award M.Sc.thesis, 93 pages.

[25] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005), Oakland, CA, USA, May 2005.

[26] Polymorphic Code http://en.wikipedia.org/wiki/Polymorphic_code

[27] Mark Stamp, D. Lin, Hunting for undetectable metamorphic viruses, Masters Thesis, Department of Computer Science, SJSU, Spring 2010.

[28] Mark Stamp, W. Wong, Analysis and detection of metamorphic computer viruses. http://www.cs.sjsu.edu/faculty/stamp/students/Report.pdf